NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

F78095

UNIX BASED PROGRAMMING TOOLS
FOR LOCALLY DISTRIBUTED
NETWORK APPLICATIONS

by

William C. Frank

December 1987

Thesis Advisor: M. J. Zyda

Approved for public release; distribution is unlimited

T238915

UNCLASSIFIED
URITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

REPORT SECURITY CLASSIFICATION Unclassified	1b RESTRICTIVE MARKINGS
SECURITY CLASSIFICATION AUTHORITY	3 DISTRIBUTION / AVAILABILITY OF REPORT Approved for public release; Distribution is Unlimited
DECLASSIFICATION / DOWNGRADING SCHEDULE	
PERFORMING ORGANIZATION REPORT NUMBER(S)	5 MONITORING ORGANIZATION REPORT NUMBER(S)
NAME OF PERFORMING ORGANIZATION Naval Postgraduate School	6b OFFICE SYMBOL (If applicable) Code 52Zk
7a NAME OF MONITORING ORGANIZATION Naval Postgraduate School	
ADDRESS (City, State, and ZIP Code) nterey, CA, 93943-5000	7b. ADDRESS (City, State, and ZIP Code) Monterey, CA, 93943-5000
NAME OF FUNDING / SPONSORING ORGANIZATION	8b OFFICE SYMBOL (If applicable)
9 PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER	
ADDRESS (City, State, and ZIP Code)	10 SOURCE OF FUNDING NUMBERS

PROGRAM ELEMENT NO	PROJECT NO	TASK NO	WORK UNIT ACCESSION NO

TITLE (Include Security Classification)
IX BASED PROGRAMMING TOOLS FOR LOCALLY DISTRIBUTED NETWORK APPLICATIONS(u)

PERSONAL AUTHOR(S)
ank, William C.

TYPE OF REPORT	13b TIME COVERED	14 DATE OF REPORT (Year, Month, Day)	15 PAGE COUNT
ster's Thesis	FROM _____ TO _____	1987 December	105

SUPPLEMENTARY NOTATION

COSATI CODES			18 SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	Distributed Programming; Local Network Tools; UNIX IPC Tools

ABSTRACT (Continue on reverse if necessary and identify by block number)
he Graphics and Video Laboratory of the Department of Computer Science has growing need for easy to use programming tools in support of distributed ocessing applications. The most pressing need is for software on three IX-based workstations connected via Ethernet. The remote interprocess mmunication tools that UNIX provides for using Ethernet are effective but nplicated to learn. This requires researchers to spend much of their time coming proficient with them instead of concentrating on the distributed plication at hand.

his work presents the design and implementation of several programming ols that allow programmers to establish and experiment with distributed ograms in the graphics laboratory environment. The tools allow a higher vel of abstraction for remote interprocess communications and establish a raightforward method for implimenting distributed programs.

DISTRIBUTION / AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	21 ABSTRACT SECURITY CLASSIFICATION Unclassified
NAME OF RESPONSIBLE INDIVIDUAL rof. M.J. Zyda	22b TELEPHONE (Include Area Code) (408) 646-2305
	22c OFFICE SYMBOL Code 52Zk

FORM 1473, 84 MAR — 83 APR edition may be used until exhausted. All other editions are obsolete

SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

Approved for public release; distribution is unlimited.

UNIX BASED PROGRAMMING TOOLS FOR LOCALLY DISTRIBUTED NETWORK APPLICATIONS

by

William C. Frank
Lieutenant, United States Navy
B.S., U. S. Naval Academy, 1981

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL

December 1987

ABSTRACT

The Graphics and Video Laboratory of the Department of Computer Science has a growing need for easy to use programming tools in support of distributed processing applications. The most pressing need is for software on three UNIX-based workstations connected via Ethernet. The remote interprocess communication tools that UNIX provides for using Ethernet are effective but complicated to learn. This requires researchers to spend much of their time becoming proficient with them instead of concentrating on the distributed application at hand.

This work presents the design and implementation of several programming tools that allow programmers to establish and experiment with distributed programs in the graphics laboratory environment. The tools allow a higher level of abstraction for remote interprocess communications and establish a straightforward method for implementing distributed programs. Additionally, they support code reuseability with software templates and are modularized to be both understandable and changeable. Recommendations are made for future research and management efforts that have been highlighted by these new tools.

Thesis
E7825
c.1

TABLE OF CONTENTS

I. INTRODUCTION 7
A. BACKGROUND 7
B. PROBLEM STATEMENT 8
C. OBJECTIVES 8
II. PROBLEM DEFINITION AND REQUIREMENTS 9
A. PROBLEM DEFINITION 9
1. Current Research Needs 9
2. Existing Limitations 9
3. Need for Programming Standards 10
4. Scope of the Distributed Problem 10
B. TERM DEFINITIONS 10
C. SOFTWARE REQUIREMENTS 13
1. Network Users 13
2. Network Programmers 14
III. SOFTWARE DESIGN AND IMPLEMENTATION 15
A. Methodology 15
B. Network Addressing 15
C. Global Database 17
D. Interprocess Communication Design 18
E. First Phase IPC Implementation. 19
F. Network Controller and Second Phase IPC Implementation 25
G. Network Server 28
H. Node Template Design 29
IV. PROGRAMMING OBSERVATIONS AND NETWORK TESTING 31
A. SHARED MEMORY AND SOCKET STREAM RESTRICTIONS 31
B. MINIMIZING DATA TRANSFER 31
C. INCORPORATION OF SEMAPHORES 32
D. REMOVAL OF ACCUMULATED SHARED MEMORY SEGMENTS 32
E. GLOBAL DATABASE STORAGE 33
F. NETWORK TESTING 33
V. CONCLUSIONS AND RECOMMENDATIONS 35
A. CONCLUSIONS 35
B. RECOMMENDATIONS 36
APPENDIX A - NETWORK INSTRUCTIONS 37

APPENDIX B - SOURCE CODE LISTINGS 39
LIST OF REFERENCES 103
INITIAL DISTRIBUTION LIST 104

LIST OF FIGURES

1. Relationship between the network, a subnet and a node 12
2. One-way shared memory communication 20
3. Two-way shared memory communication 22
4. Network Architecture 29

I. INTRODUCTION

A. BACKGROUND

The Graphics and Video Laboratory of the Department of Computer Science has a growing number of research topics that involve distributed processing applications. The current working environment consists of three UNIX[1] based Silicon Graphics IRIS workstations connected via Ethernet. There are two reasons for seeking to coordinate remote processes within this system.

First, we wish to allow graphics simulations that represent real world scene changes as accurately as possible. This is done by updating the graphics picture as frequently as the IRIS workstation hardware permits (approximately every thirtieth of a second). However, as real time applications become more complex, the processing time required to prepare a new screen image for display increases beyond one thirtieth of a second. This means that the picture update frequency becomes less than the optimum supported by the hardware. Therefore, we would like to lower the processing time required between updates as much as possible by distributing the load over available machines on the Ethernet.

Second, we would like to support multiple workstation simulations involving interactions with several users at one time. An example of this would be a command and control simulation that allows simultaneous input from multiple workstations while keeping each one updated with the shared scenario. Additionally, the capability for

[1]UNIX is a trade mark of Bell Laboratories, Incorporated.

dynamic entry and exit from such a simulation is desirable.

B. PROBLEM STATEMENT

As researchers attempt to solve the distributed problems described above, they find that they must first become proficient with the remote and local interprocess communication (IPC) tools available in the graphics laboratory. At present, both the UNIX system features and laboratory developed tools have the following drawback: they offer either high capabilities with high learning curves, or limited capabilities with reasonable learning curves. As a result, many researchers are forced to either seek a non-distributed research topic or to narrow the scope of their efforts.

C. OBJECTIVES

The main goal of this work is to establish a firm foundation of software tools that provides programmers with high capabilities for local/remote interprocess communications. Such tools should require only a short learning curve. A more detailed definition of the problem is presented in Chapter II along with a list of corresponding software requirements. Significant highlights of the software design and implementation process are covered in Chapter III. Chapter IV gives a summary of testing results. Finally, Chapter V gives project conclusions and recommendations for further research.

II. PROBLEM DEFINITION AND REQUIREMENTS

A. PROBLEM DEFINITION

1. Current Research Needs

There are currently two real-time graphics simulation programs that local researchers are attempting to expand into the distributed environment. The first program simulates the Army's Fiber Optically Guided Missile (FOGM) system. It has become complex enough to burden a single IRIS workstation's processing resources to the point of showing a marked drop in real-time performance below the ideal graphics update threshold. The second program simulates the Naval Tactical Data System (NTDS). For both the FOGM and NTDS simulations, it is desired to run distributed scenarios from two or more workstations at once. This identifies the necessity of being able to handle simultaneous interactions on remotely distributed data sets originating from a dynamic set of user programs.

2. Existing Limitations

The laboratory developed programming tool currently being used for remote interprocess communication applications allows a reasonable level of abstraction for use by the average programmer but is limited in capability. The main function that it performs is the setting up of a communications link between two remote processes. It leaves essentially all of the work for distributed program development to the programmer with no guidelines to follow.

3. Need for Programming Standards

While the implementation of distributed programs in the graphics lab environment remains in beginning experimental stages, the approaches chosen for problem solutions vary with each topic and researcher. The danger here is that as more work is founded on an established approach, it becomes more costly to effect changes of the underlying distributed framework. Fortunately, this has not yet become a problem with any of the distributed research topics at hand, making now the perfect time to implement standards with future developments in mind.

4. Scope of the Distributed Problem

The scope of the distributed problem quickly becomes too large to treat as a subtopic of ongoing graphics projects for two reasons. First, the learning curve is too high for the average programmer to become familiar enough with UNIX-level IPC tools to develop new distributed capabilities without full attention to that topic as a research problem. Second, distributed computing involves many non-graphics related considerations, such as global database management, that make it best kept as a separate research focus. Work such as this is needed to make it easy to select a boundary between dedicated distributed computing and graphics applications. The same tools should allow a researcher to focus on either topic without much concern for the other.

B. TERM DEFINITIONS

At this point, it is necessary to define terms that properly encapsulate the main ideas drawn from the problem definition. This makes the formal software requirements easier to understand. In reality, they are based on major assumptions about the prospective *logical* structure of the desired network. They have been derived largely from classical

operating system and distributed computing examples which are mentioned briefly but are not a major focus of this work.

The *network* refers to the graphics laboratory environment of multiple IRIS workstations connected via Ethernet and pertinent processes, software and databases.

A *subnet* signifies a single distributed program within the network; more specifically, a collection of local and/or remote processes that interact with each other in any way. Subnets are unique by name, even if they are exact logical duplicates of each other.

A *node* is the basic building block of a subnet and refers to a single process that has the capacity for two way communications with any other process in the network according to a standard library of logical communications media and protocols. A node usually controls one or more physical and/or logical resources in a server relationship to other nodes (and vice versa). Nodes are uniquely named throughout the network from a combination of the name of the subnet that they belong to and their given name for use within that subnet.

A *node template* is the program from which a node process is instantiated. Each node template is unique by name within the network and the entire library of them is usually replicated over all workstations in the network.

A *subnet template* consists of a stored list (replicated over the network) of: names of node templates to be used, a corresponding list of unique names for each related node to assume within a subnet, and various initialization instructions to establish node communications and start planned interactions. Subnet templates are unique by name.

An *abstract module* is the central building block of a node template and is the software implementation of an abstract data type and/or physical resource control. It is

intended to allow a programmer to use the abstract data type or physical resource with minimal knowledge of the underlying software. Correspondingly, abstract modules are made with minimal assumptions about their external usage environment. Different abstract modules are unique by name and a library of them is usually replicated over the network.

Figure 1 shows a pictorial representation of the relationships between several of the above terms. In order to more fully grasp the main ideas behind these terms, a layman's description of how the network could have reached the state shown in Figure 1 is given here. First, subnet 'alpha' was created by using a subnet template called 'server_one_client_two' that contains the following information:

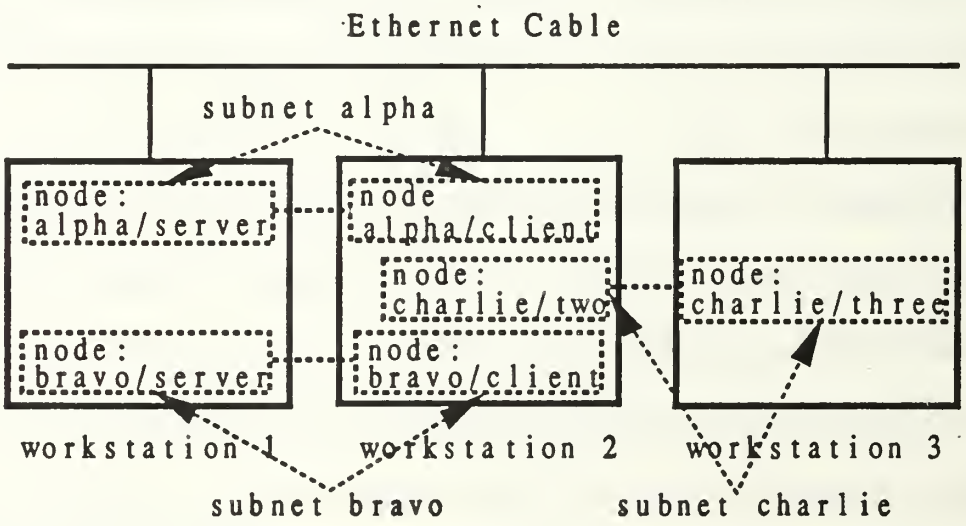

Figure 1. Relationship between the network, a subnet and a node

- Start a node called 'server' on workstation 1 using the node template called 'server'.
- Start a node called 'client' on workstation 2 using the node template called 'client'.
- Have the node 'client' establish communications with the node called 'server'.

The subnet 'bravo' was then created with the same subnet template but is distinct by name. Subnet 'charlie' was created by using a subnet template called 'comms_two_three' that contains the following information:

- Start a node called 'two' on workstation 2 using the node template called 'two_way_comms'.
- Start a node called 'three' on workstation 3 using the same node template.
- Have node 'two' establish communications with node 'three'.

C. SOFTWARE REQUIREMENTS

Software requirements can be broken up into two major areas according to the beneficiaries of the work as listed below.

1. Network Users

We want to allow a network user to view current subnet data, activate a new subnet from a selected subnet template, or terminate a subnet. By differentiating between a subnet and a subnet template, more than one instantiation of the same subnet template can be active at the same time under unique subnet names.

We want to allow a network user to quickly and easily create, display, modify, store and delete subnet templates. By providing differentiation between a library of node templates and the actual nodes that are to result from the use of a subnet template, the same node template can be used more than once within the same subnet template.

2. Network Programmers

We want to provide network programmers with abstract local/remote IPC modules that combine high capabilities with a short learning curve. Guidelines are also needed for creating and storing abstract modules and resource templates for use within the network. A related need is for easy access to a dynamic and distributed database useful for network applications.

III. SOFTWARE DESIGN AND IMPLEMENTATION

A. Methodology

A bottom up approach was chosen for the software design phase of this work. This decision was made after noting that the structure of most software modules could not be approached without an assurance of the underlying interprocess communication tools on which they would rely. After these tools were provided, it was possible to divide top level design issues into functionally partitioned modules that could each be implemented with an isolated focus of attention.

Although the bottom up approach made design and implementation efforts more efficient in this case, it was not expected to result in the best logical architecture for the completed system. The result of this work was intended to be a working prototype that could then be modified with a top down approach, with a second generation of the system suitable for widespread use.

As each new topic is presented in this chapter, its development is followed through from design to implementation. This allows each concept to be grasped with a complete and unbroken chain of thought. In turn, each succeeding topic builds upon those before it to give a smooth picture of the actual evolution of the system.

B. Network Addressing

The first design step was to establish network addressing standards to be used throughout the system. Every node/process within the network has a unique logical and physical address. The logical address determines a node's subnet membership and

participation, and is independent of the node's physical location. A node must also have a physical address associated with the logical address to provide for direct interprocess communication. The physical address must be able to describe a unique process on a unique worksation within the network.

Since a logical address contains information that gives the user a clear picture of the logical relationships between nodes, it was implemented in a manner to facilitate user readability. It is represented by an ASCII character string consisting of the name of the subnet that the node is participating in followed by the node's given name within that subnet. For clarity the two parts of the address are separated by a "/".

Physical addresses were implemented in a manner more readily useable by system programmers. They are stored as 4-byte long integers that include the UNIX assigned process identification number and a network unique number identifying the workstation that the node actually resides on. The actual bit assignments are as follows:

- bits 0 to 12 - set to zero
- bits 13 to 27 - UNIX process identification number
  (0 to 29,999)
- bits 28 to 30 - workstation number (1 to 7 with 0 unused)
- bit 31 - (sign bit)

The reason bits 0 to 12 are not used in physical node addresses is to allow the addresses to be used as part of an interprocess message's identification. By continually incrementing within these bits as messages are sent, a node can insure unique message identification that also carries its physical address as the message origin.

C. Global Database

At this point, the need for a global network database was recognized. Its primary function is to store physical and logical node addresses in a manner that allows them to be linked together. The most beneficial use is the ability to retreive an unknown physical node address based on the known logical address. To allow global use, the database must be replicted on all workstations within the network. A single storage or deletion operation must be performed in duplicate on all workstations as close to simultaneously as possible. Using this approach makes all data available through local retrieval operations. This gives the added benefit of dealing with one information source.

For implementing the global database, the UNIX "dbm" facility was chosen as the storage medium. By using a special hashed file structure, dbm allows ASCII string records to be stored in association with ASCII string keys. Records can then be retrieved or deleted if their key is known.

In order to complete the facilities of the global database, a method of distributed control for all database modifications is needed. Its main purpose is to prevent the simultaneous execution of incompatible commands. An example of an incompatible command is an attempt to simultaneously store different records under the same key. Actually, the dbm facility is more restrictive than this. It cannot handle any concurrent database modification operations on a particular hashed file. Unfortunately, the necessary distributed controls are not yet implemented and it is up to the user/programmer to sequentially manage database modifications. The above restrictions do not apply to database retrieval operations.

D. Interprocess Communication Design

Interprocess communication tools were needed at this point before any other software modules could be designed. This was due to uncertainty about how the underlying UNIX V tools [1] might effect the top level IPC interfaces. The IPC design problem was divided into two phases. The first phase was to design local IPC tools and the second phase was to expand the design to handle remote IPC. The idea behind this approach was to allow all IPC to be treated as local IPC at the top level of usage while keeping remote IPC transparent. Software modules that use the IPC tools can then use node addresses without concern for where the corresponding node is actually located.

The top level IPC interfaces were designed as generically as possible to allow portability of the software that uses them. This was the approach taken by Rochkind [2] whose local 'send' and 'receive' operations keep the same format through several UNIX implementation examples. The IPC interfaces chosen in this work expand those basic 'send' and 'receive' capabilities to allow the selection of protocols that can be either synchronous or asynchronous in nature. These protocols are described in detail by Cooper and Kearns [3]. A unique version of them is built into the IPC interface as follows:

- A 'net_recv' operation allows a node to wait for an incoming message from a specific node or any node.

- A 'net_poll' operation is similar to 'net_recv' except that it terminates if no incoming message is present for immediate reading.

- A 'net_send' operation allows one of three choices before a message is sent:
  - Terminate the operation if the destination node has not

yet received the preceding message from the sending node.
- Block until the preceeding message is acknowledged.
- Send the new message without waiting.

After the message is actually sent, 'net_send' allows one of two choices:
- Wait for the new message to be acknowledged.
- End the operation without waiting.

Message acknowlegdement was designed to have broader implications than the traditional guarantee of receipt. In this work it signifies the final disposition of the action that the message was to take. This way, a node can leave a message unanswered until it is actually ready to carry out the associated actions. In such a system it appears to the sender that it is not yet received. The associated operations are 'acknowledge' and 'negative acknowledge', which take a single message ID as input. A positive acknowledge is indicated by writing the actual requesting message ID value in the appropriate acknowledgement buffer. A negative acknowledge is signified by a negation of this value.

E. First Phase IPC Implementation

The low level implementation tools for local IPC were chosen for speed. The fastest method that UNIX System V (the flavor of UNIX on the IRIS workstations) provides for interprocess communication within a single workstation is through the use of shared memory. The related UNIX facilities, although somewhat difficult to learn, allow processes to read and write to common segments of memory. This means that large amounts of data can be exchanged between processes by reference rather than by copying. Shared memory is the only method that offers this benefit. Unfortunately, it is only available in UNIX V as opposed to all other versions of UNIX.

With shared memory, there are two types of communication that are used in the network. The first of these is for a one-way transfer of information. This applies when a serving process continually updates a data structure stored in shared memory and has no need for feedback from a client process about how the data may have been read and used. Figure 2 shows how a segment of shared memory can be used in this way and is described here. The 'write' algorithm to be used by node 'A' when updating data 'X' is as follows:

- Set data structure 'X' status to -1 to indicate a transition.
- Perform updates.
- Set data structure 'X' status to a new identifying number.

The 'read' algorithm to be used by a node 'B' when using data structure 'X' is:

- Read data structure 'X' status and cancel this operation if the value is -1 or unchanged when only new data is sought.
- Read data structure 'X'.

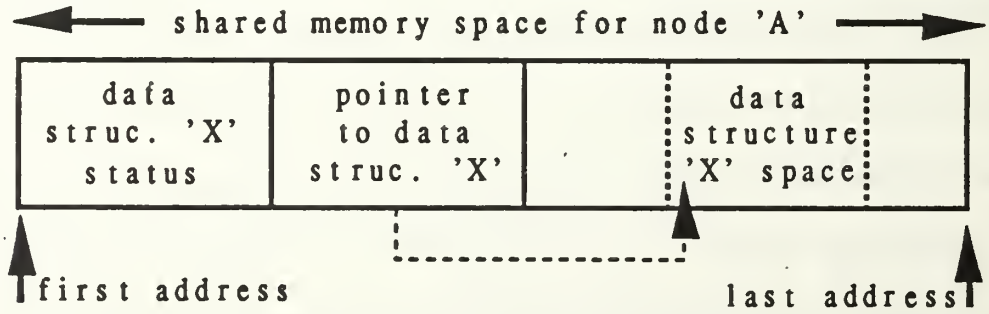

Figure 2. One-way shared memory communication

- Read data structure 'X' status again and cancel the effects of this operation if it has changed since the first step.

Although this algorithm can have a low overhead when compared to two-way coordination, care must be taken to insure that a data structure is left in an unchanged state by the writing process long enough to let prospective reading processes capture it intact.

The second type of shared memory communication used is two-way message passing over established logical links between two processes. Each process places an outgoing message in an assigned shared memory ring buffer directly after the slot used for the preceeding message. This buffer is written to only by the process it is assigned to but may be read by any other process. The reason it is called a ring buffer is due to its use as a circular queue. When a new message has the potential for overwriting the upper boundary of the ring buffer, it is simply placed at the lower boundary and overwrites old messages remaining there. Messages of variable length can be sent as long as they fit within the ring buffer. Receiving processes must be advised of messages that are sent to them and where they are actually located within the sender's ring buffer. They must also be given the opportunity to relay an acknowledgement back to the sender. To provide for this, each process maintains three buffers for each process it intends to have two-way communications with. The first buffer contains the message ID of the last message sent to the opposite process. The second buffer contains a pointer to that message's location within the ring buffer. The third buffer contains the message ID of the last message sent by the opposite process that was also acknowledged by the local process. When combined with the use of three corresponding buffers owned by the opposite process, a

two- way communication link is defined. Figure 3 shows the use of shared memory for this purpose and is explained here. The following algorithm shows how node 'A' sends a message to node 'B':

- If node 'B's 'last message ID received' is not equal to node 'A's 'last message ID sent' the options are:
  - cancel this operation.
  - wait for node 'B's acknowledgement before proceeding.
  - proceed without waiting.

- Set node 'A's 'last message ID sent' to -1 to indicate a transition.

- Write the message at the next available slot in node 'A's ring buffer.

- Set node 'A's 'last message ID sent' to the ID of the new message.

- End this operation or wait for node 'B's acknowledgement of the new message.

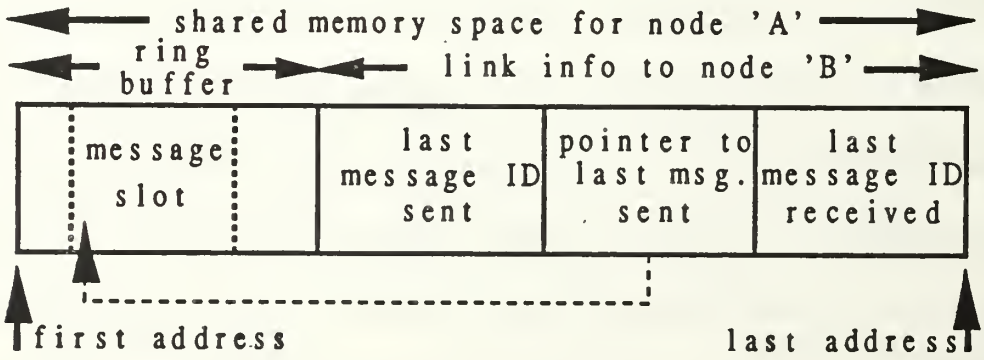

Figure 3. Two-way shared memory communication

Node 'B' reads new messages from node 'A' with this algorithm:

- If node 'A's 'last message ID sent' does not indicate a new message, the options are:
  - end this operation.
  - wait until a new message is sent.

- Retain the message pointer for reference or make a copy of the message if long term retention is desired.

The main disadvantage of the algorithm above is that it remains possible for the sending node to overwrite a message before it is received by the intended destination node. Currently, the only practical solution to this problem is to enlarge the ring buffer as necessary.

Each node in the network is actually assigned 2048 bytes (an experimental size) for both methods of communication. The structure of the complete segment is as follows:

- first 512 bytes - data structure storage for one-way communications (referred to as data space).
- next 1024 bytes - a single ring buffer for messages.
- next 128 bytes - room for 10 links to other nodes (referred to as link space).
- last 384 bytes - available for experimental use.

For each of the 10 possible two-way links within a node, there is a data structure (separate from shared memory) that encapsulates the information necessary for its use. This includes the following:

- the physical address of the opposite node (if link in use)
- pointers to the local shared memory location for:
  - 'last message ID sent'
  - 'message pointer'

- 'last message ID received'
- pointers to the related shared memory locations for the opposite node.
- various information to allow 'navigating' through the opposite node's shared memory segment.

A message buffer data structure is provided for use in sending and receiving operations. It serves to keep the following related information available within a single source:

- the message identification number.
- a pointer to a copy of the message ID within the message slot in the ring buffer to allow for a check on the integrity of the message (it could be overwritten).
- a pointer to the actual message data within the message slot in the ring buffer.
- a pointer to a message data size value within the ring buffer.
- a pointer to a message type value within the message slot in the ring buffer.
- room for optional 'hard copies' of the message slot data above.
- any experimental message parameters.

A 'make message' operation is provided to make the use of a message buffer data structure easier when creating a new message. It uses the following input parameters:

- a character pointer to the actual message data.
- the size of the data.
- the message type.
- a pointer to the empty message buffer to be used.
- a message mode selection to choose weather or not additional 'hard copy' information should be stored within this message buffer.

The result is a message buffer complete with a newly assigned message ID and proper data placement within a shared memory message slot.

F. Network Controller and Second Phase IPC Implementation

At this point, the need for a network controller process on each workstation is recognized. The functions of the network controller are to: (1) assign shared memory segments to and establish communications with newly created nodes upon request; (2) forward various messages between unconnected nodes; and (3) establish and control interworkstation communication and make it appear as local communication to local nodes. The network controller can be considered as a special node that remains active at all times and is an implied member of every subnet when viewed by a local node. Its logical address is "net/cntrl/workstation_name".

Upon startup, the network controller obtains one large shared memory segment that is logically divided up into 16 segments, each with the node shared memory structure described earlier. The first segment belongs to the network controller itself, the next 5 are allocated for local node use, and the last 10 are allocated for remote node interfaces. When a new node becomes active and desires to obtain its shared memory segment, it follows the algorithm listed below:

- Write its physical node address into the first bytes of the network controller's data storage area of shared memory (this is the only buffer in shared memory that has uncontrolled write privileges for all new nodes!).

- Read a succeeding buffer to see if the controller has acknowledged by duplicating the requesting address there; if not then return to the first step.

- If a negative acknowledge was received then there are no more unused segments available and the requesting node must terminate.

- Read other buffers to obtain segment assignment information.

- Complete segment initialization and acknowledge readiness to the controller through another buffer in the controller's shared memory.

- Use link zero for all succeeding communication with the network controller.

The corresponding network controller side of the algorithm is listed here:

- If a new node has not yet acknowledged readiness of its assigned segment then end this operation.

- If an unattended node has written in the request buffer then acknowledge it and place assigned segment information in the appropriate buffers; or, negatively acknowledge if no more segments are available.

After a node has established communications with the network controller, it can then establish direct two-way links with other nodes as described earlier. This is a two step process where (1) the requesting node sends a request to the accepting node through the network controller with pertinent connection information, and (2) the accepting node returns a reply through the network controller with similar information. A node can refuse link requests if all 10 of its links are currently in use.

For interworkstation communication, the only truly reliable method that is currently implemented in UNIX is the use of Internet stream sockets. The related facilities are difficult to learn, even with the extensive instructions available in reference [4]. It is the network controller's job to establish one of these remote links with each of its counterparts within the network. By using only these links for all interworkstation network communication, the overhead of numerous remote connections between nodes can be avoided.

The network controller bridges the gap between local and remote communication. First, it intercepts all link requests to remote nodes and, by coordinating with the net controller on the remote workstation, makes an exact duplicate of the remote node's shared memory segment in a local segment allocated for this purpose. The remote workstation does the same for the local node. Both segments are routinely updated with the minimum amount of data transfer necessary. The ring buffer and link space of a shared memory segment that represents a remote node need only contain messages destined for the workstation on which it is located. Also, the data space of the segment need only be updated when changes occur as discerned using the one-way communication method described earlier. *Note: The local/remote IPC interface within the network controller is not yet implemented.* Remote messages can be sent via the network controller with a temporary 'forward' operation that is provided.

Lastly, a special algorithm must be followed when starting all of the network controller processes during network initialization. This is a point where the lack of global database control must be worked around in order to advertise interworkstation addresses. It must be a certainty that only one network controller at a time will modify the database during this initialization. Also, the network must be able to be started from any workstation and in a completely deterministic manner. The approach that the algorithm takes can also be used when starting up a subnet of several nodes. The main steps are listed below:

- In increasing order of workstation ID numbers, wait for the workstations with numbers less than the local one to advertise interworkstation addresses.

- In the same order, initiate connections with the corresponding remote network controllers.

- Initialize and advertise the local interworkstation address.

- In increasing order of workstation ID numbers, accept connection requests from the corresponding remote network controllers.

G. Network Server

A network server is needed as a second type of permanent network process on each workstation. Its main purpose is to perform all network requests that can take so much time to complete that they would cause the network controller to suffer a serious drop in performance if it had to handle them. The network controller must be free to update remote shared memory segments as frequently as possible. All global database operations are too slow in this respect and must be performed by the network server. Subnet startups and other user requests also fall under this category. The list below summarizes these functions:

- global database write operations.
- global database deletion operations.
- global database read operations (although any node may perform them directly).
- subnet startup requests.
- subnet termination requests.
- network status requests.

The network server can be viewed as a permanent node that is reached through access to the network controller. It has the logical address, 'net/serv/workstation_name'. The network architecture with both the network controller and network server processes is

shown in Figure 4. *Note: The network server was not implemented in this work.* A program called 'netaccess.c' was provided to show how a user interface to the network server can be implemented.

H. Node Template Design

The design criteria of a node template is similar to Hoare's implementation of a monitor [5] (functionally speaking). A monitor is a coded algorithm that is designed to provide controlled access to specific logical and/or physical resources and serve only one client process at a time. This applies to a node template because the network server prevents identical nodes from being instantiated from a single node template at the same time. A monitor queues requests so as not to starve any requesting process. This is effected in a node template by serving active communication links in a rotating fashion. The equivalent of Hoare's waiting (queueing a request) operation is represented by a

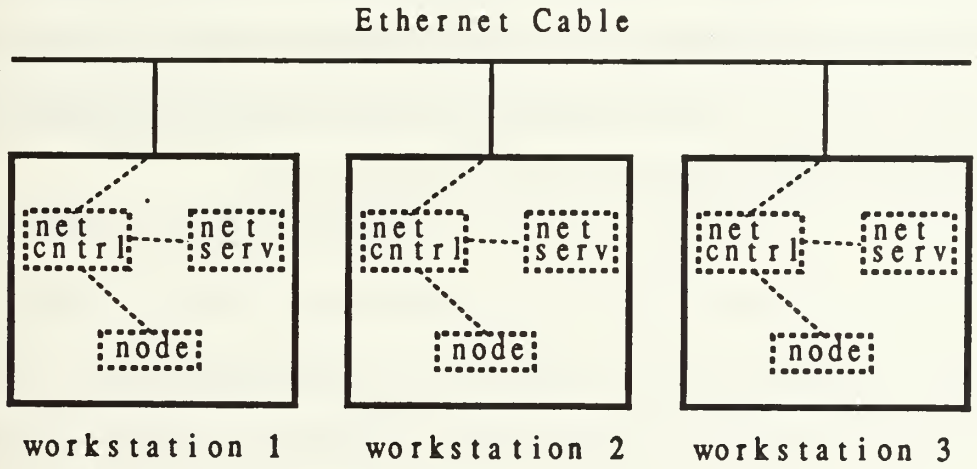

Figure 4. Network Architecture

node template's discernment to not acknowledge an incoming message when it cannot yet handle the request completely. A practical example of this is the net controller delaying the acknowledgement of a message forwarding request until the link to the destination node indicates readiness to receive the message. The equivalent of Hoare's signaling (accepting a request for action) operation is a polling of active links. In handling requests as a server, a node template can in turn require the services of other nodes as a client. A node template can also provide for predefined resource controls without an external request. In fact, this is often the most important function of a node template, which distinguishes it from a monitor.

The major building blocks of a node template are abstract data modules, which are the implementations of abstract data types. They follow modularization and information hiding principles as highlighted by Parnas [6]. Abstract data modules intended for network use differ from traditional abstract modules in one way. They allow for data storage to be assigned by the processes that use them rather than by the operating system. This allows data to be stored and modified within the shared memory segments when one-way communication is desired. The network controller program, 'netcntrl.c', is a good example of the implementation of a node as described above.

IV. PROGRAMMING OBSERVATIONS AND NETWORK TESTING

A. SHARED MEMORY AND SOCKET STREAM RESTRICTIONS

During the testing and debugging phase of this work, several key lessons were learned concerning the UNIX tools used to implement the network system. They are highlighted here in the interest of saving time and effort for follow on research. The most important observation pertains to both shared memory and socket facility usage. Individual shared memory modifications and socket transfer operations must be done with 4 byte increments. For instance, if a single 2 byte integer is to be stored in shared memory or sent over a socket connection, it must first be converted to a 4 byte integer. (e.g. The result of an attempt to send two, 2 byte integers over a socket connection in sequence, will result in an incorrect value at the destination if read in the same order and format that they were sent in.) Although not confirmed in avaiable documentation, it is suspected that any restriction to 4 byte increments is related to the 4 byte word size of the Motorola 68020 processor used as the IRIS workstation CPU.

B. MINIMIZING DATA TRANSFER

Also related to socket communications is the need to keep the number of bytes transferred between workstations as low as possible. One way to do this is to avoid the conversion of numbers to ASCII form before transferring. Although socket operations must be performed on data in a character string format, a number can be copied byte by byte into a character string without conversion to ASCII form. This can lower the number of bytes needed for a character string representation of a number by

approximately two thirds. For example, an 11 character ASCII representation of an integer value can be directly represented with a 4 byte value. It should be noted that character string lengths must be in 4 byte increments, as the same storage and transfer restrictions apply as mentioned above.

C. INCORPORATION OF SEMAPHORES

The network performance can be improved in another way besides minimizing the data flow between workstations. By adding the use of UNIX semaphore operations, busy waiting can be removed from several places within the current network tools. UNIX semaphores are very difficult to learn but give the advantage of atomic operations on whole sets of semaphores at a time. An example of a prospective use of set semaphores is to allow a set to represent all communications links that a process has activated. When the process wishes to block until a message is received over any link, it can do so with the standard set semaphore operations provided by UNIX V. A corresponding sending process would then perform a standard 'V' operation on its assigned semaphore within the set to advise the receiving process that a message has been sent. UNIX V implemented semaphores are much faster than the busy waiting used in the current implementation of the network for similar message passing operations.

D. REMOVAL OF ACCUMULATED SHARED MEMORY SEGMENTS

Another area of concern is the wasting of memory space by accumulating abandoned shared memory segments. This is the case when processes are terminated without first removing shared memory space that they created. Although this situation cannot be

completely avoided during the experimental stages of a new shared memory system, it can be easily corrected. A short program called 'rmvshmids.c' (remove shared memory IDs) is provided to clear the system of all shared memory segments that have been created by the user's processes. This program need not be run whenever an unused segment remains behind its implementing process; just when the total memory in the accumulated segments becomes large enough to affect performance. In any event, it should always be run after a programming session is completed to leave a 'clean' system for others.

E. GLOBAL DATABASE STORAGE

The next programming recommendation concerns the use of UNIX dbm facilities. When storing an ASCII string as either a key or a record, it is best to indicate that the string sizes are one more than the actual number of characters whenever dbm procedure calls are used. This will cause the string's null terminator character to be stored or retrieved as well, which allows non-dbm read operations to use dbm hashed files directly (once pointer references are known).

F. NETWORK TESTING

A subnet was created to demonstrate the speed limitations of the network under the most time-consuming communication conditions. It consists of a simple, mouse-controlled, line drawing program as the type of node within the subnet and is run on any or all of the three workstations in the network. Depending on user selections given during each node's startup, line drawing with the mouse on one workstation is also displayed on

the other workstations. Mouse data is transferred over local shared memory and remote socket links with message acknowledgements waited for before new messages are sent. The combination of frequent messages from a moving mouse and the short message size for the mouse data (32 bytes) leaves the actual communication protocol as the largest overhead factor affecting performance.

The results from tests of the subnet described above (see Appendix A for instructions) showed through visual evaluation that the system was too slow to maintain a smooth graphics picture with the chosen protocol. However, the speed was sufficient for subnets that only require performance that is close to real-time (i.e. approximately one second response time). The communication protocol required to test real-time graphics performance is the use of one way shared memory/socket communication. Unfortunately, the necessary coupling between shared memory and socket communication was not implemented in this work.

V. CONCLUSIONS AND RECOMMENDATIONS

A. CONCLUSIONS

The design of a locally distributed network system has been presented for use on UNIX V based workstations connected via Ethernet. It offers high level interprocess communication facilities without requiring programmers to become proficient with low level UNIX tools. This allows research efforts involving locally distributed programs to be concentrated on higher level topics at hand. Conversely, the network system also offers a foundation for future research focused specifically on the details of distributed program control.

A prototype implementation of the network's global database and interprocess communication facilities has been provided for use on the Silicon Graphics IRIS workstations of the Graphics and Video Laboratory. The global database facilities offer global store, retrieve and delete operations through a high level interface to UNIX dbm facilities. The network IPC facilities combine UNIX V shared memory and socket stream facilities to provide high level, synchronous or asynchronous communication between remote and/or local processes.

A distributed graphics program was developed to test the implemented network tools by allowing various configurations of remote mouse control between IRIS workstations. Testing showed that the current network system's configuration is sufficient for near real-time programs that can operate with a one second response time for remote IPC operations. Further enhancements of the prototype network system are necessary before

real-time distributed graphics programs can be handled without a marked decrease in performance.

B. RECOMMENDATIONS

As the primary follow on research topic, a review of the network system design with a top down approach is recommended. This second generation of the network will allow a broader perspective to be used when reviewing the capabilities and interfaces to be included in the tools provided. Accordingly, this will result in a network architecture that is more suitable for widespread use than the current prototype implementation. Although sufficient IPC capability exists in the prototype, it is *not* recommended for other than testing purposes.

Within the second generation of the network system, the network server process should be added to provide a central management point for user interface to the network. This process can provide the user with a quick and easy means for manipulating network resources and can relieve the network controller process of all requests not directly related to its IPC duties.

Also to be completed in the second generation of the system is the bridge between local shared memory communication and remote socket communication. This will allow a full implementation of the one-way shared memory communication described in this work and the capability to handle real time distributed graphics programs.

Lastly, the network system's performance can be greatly enhanced by eliminating busy waiting from several key points within the implemented tools. This can best be done with UNIX semaphore facilities. They are difficult to learn and use, but offer the advantage of atomic operations on whole sets of semaphores.

APPENDIX A - NETWORK INSTRUCTIONS

A. GLOBAL DATABASE OPERATIONS

The global database may be maniplated directly by a network user by using appropriate commands at the UNIX system prompt. All database commands have the following format:

dbcntrl database_name command command_scope key_string record_string

Specific commands use the last three arguements only as necessary and are listed below for the current network configuration.

dbcntrl net s g key_string record_string
- "Store 'record_string' under 'key_string' in hashed file 'net' on all workstations in the network (globally)."

dbcntrl net f l key_string
- "Fetch the record string stored under 'key_string' in the hashed file 'net' on the local workstation."

dbcntrl net d g key_string
- "Delete the record string stored under 'key_string' in the hashed file 'net' globally."

dbcntrl net c g
- "Create the database (hashed file) 'net' globally."
- Note: Any previous implementation of 'net' is erased.

B. SETTING UP THE NETWORK

The network can be started by following the procedure listed below.

(1) Execute: "dbcntrl net c g" or "dbcntrl net d g netcntrl/iris1"
+ "dbcntrl net d g netcntrl/iris2"
+ "dbcntrl net d g netcntrl/iris3".

The purpose of this step is to clear the database of any old network addresses that might be left from a previous run of the network.

(2) At each workstation (use the three side terminals) type "netcntrl &".

(3) Press 'carriage return' on all workstations within a few seconds of each other.

(4) Wait for the "on line" message to appear at each workstation (three to ten seconds).

C. RUNNING THE TEST PROGRAMS

The subnet of "mouse" programs can be run to test the network by following the steps listed below:

(1) On one workstation (side terminal) type "netdraw" follwed by an optional list of one or two remote workstation names (e.g. "netdraw iris1 iris3" typed on iris2). Press carriage return. The process will wait until a netdraw program is started on the indicated remote workstations and then start a simple line drawing procedure. Mouse control on the local workstation will then be duplicated on the others indicated.

(2) Startup similar programs on the other workstations.

(3) Exit programs by pressing the middle mouse button on the local workstation.

(4) For each workstation that ran a netdraw program, delete the record stored under the key "netdraw/workstation_name" globally. Do this before succeeding runs on the same workstation.

```
/*****************************************************************

    This is a UNIX V program called netcntrl.c.
    It is part of thesis work done by Lt. W. Cory Frank during summer and
    fall quarter 1987 under the title, "UNIX Based Programming Tools
    for Locally Distributed Network Applications."
    The main function of this program is to function as the central
    part of a distributed network program. It is replicated over all
    workstations participating in the logical network.

*****************************************************************/

#include "netglobal.h"
#include "netlocal.h"
#include "netipc.h"
#include "netdb.h"
#include "netmsgtypes.h"
#include <bsd/sys/time.h>
#include <bsd/sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <bsd/netdb.h>
#include <stdio.h>

/*** All of the following external variables are from netipc.c
***/
extern shd_mem_link_info link[SHD_MEM_MAX_LINKS];
extern long local_node_addr,
        cntrl_node_addr,
        shd_mem_first_addr;
extern int  cntrl_shd_mem_id,
        cntrl_segment_offset,
           local_segment_offset,
           high_link_id;
extern char *local_segment_first_addr;

/*** These static variables are global to netcntrl.c only
***/
static inter_host_info   remote_link [NUMBER_OF_HOSTS + 1];
static int max_sock_num = 0;
```

```
main (argc, argv)
  int  argc;
  char *argv[];
{
  /*** establish the physical node address for the local process
  ***/
  local_node_addr = (HOST_NUMBER * 0x10000000) + (getpid() * 0x2000);


  /*** one workstation can start the remote net controllers but here
       but this is not recommended as long as the network is in an
       experimental phase and fuul of information feedback provisions.
       What often happens is that a huge volume of feedback to the
       originating terminal overloads Ethernet and causes unpredictable
       and sometimes catastophic results. The current alternative used
       here is to type the command lines on all workstations and push
       <carriage return> on all within seconds of each other.

  start_remote_net_controllers (argc);
  ***/


  /*** This is a fully deterministic, global startup algorithm designed
       to prevent more than one process at a time from changing
       the global database while assuming connection responsibilities
       among each other.
  ***/
  connect_to_remote_net_controllers ();


  /*** initialize the single shared memory segment used for IPC by the
       local workstation.
  ***/
  shd_mem_init ();


  /*** not yet implemented
  start_local_net_server ();
  ***/


  printf ("%d: %d: This net controller is now in service **************0,
          HOST_NUMBER, getpid());
```

```
  /*** control loop
  ***/
  while (1 == 1)
  { /*** Check remote sockets for incoming messages from
          other workstations.
    ***/
    poll_remote_connections ();


    /*** Check local nodes for messages to the net controller.
    ***/
    poll_local_connections ();


    /*** not yet implemented
    update_remote_shd_mem ();
    ***/


    /*** Check for and new service local nodes requesting tp establish IPC
          with the net controller.
    ***/
    service_net_connection_request ();
  }
}



/*** This procedure starts corresponding network controllers on
    all other hosts in the logical network. It uses remote system
    shell commands and adds a dummy command line argument to
    signify a secondary process. If this procedure is used in a
    secondary process it is a null operation. Note: if this type
    of remote startup algorithm is modified improperly it could
    result in an exponential explosion of processes on the network
    hosts!
***/
start_remote_net_controllers (argc)
  int argc;
{
  int  count,
          marker;
  char hosts[128],
          next_host_name[16],
       syscmd[64];
```

```
  if (argc > 1)
  {  /*** This is a secondary process
     ***/
     return (0);
  }

  strcpy (hosts, HOSTS_IN_ORDER);
  marker = 0;
  for (count = 1;
         count <= NUMBER_OF_HOSTS;
         ++count)
  {  for ( ; hosts[marker] == ' '; ++marker){}
     sscanf (&hosts[marker], "%s", next_host_name);
     for ( ; (hosts[marker] != ' ') && (hosts[marker] != '  '); ++marker){}
     if (count != HOST_NUMBER)
     {  strcpy (syscmd, "rsh ");
        strcat (syscmd, next_host_name);
        strcat (syscmd, " '/work/frank/network/netcntrl xxxxx' &");
        system (syscmd);
     }
  }
}


connect_to_remote_net_controllers ()
{
  int  count,
         marker,
         length;
  char hosts[128],
       key_string[32],
       content_string[32];
  struct sockaddr_in server;
  struct hostent *host_ptr,
                 *gethostbyname();

  /*** Obtain all the host names in the logical network and initialize
         associated message acknowledgement parameters.
  ***/
  strcpy (hosts, HOSTS_IN_ORDER);
  marker = 0;
  for (count = 1;
         count <= NUMBER_OF_HOSTS;
         ++count)
  {  for ( ; hosts[marker] == ' '; ++marker){}
     sscanf (&hosts[marker], "%s", remote_link[count].host_name);
```

```
  for ( ; (hosts[marker] != ' ') && (hosts[marker] != '  '); ++marker){}

  remote_link[count].sent_msg_id = -1;
  remote_link[count].ack_msg_id = -1;
}


/*** Initialize the connection media to all remote hosts with
       network identification numbers less than the local host's
       before actual connections are attempted.
***/
for (count = 1;
       count < HOST_NUMBER;
       ++count)
{  /*** create Internet stream sockets
  ***/
  if ((remote_link[count].socket =
          socket (AF_INET, SOCK_STREAM, 0)) < 0)
  {  printf ("%d: %d: ", HOST_NUMBER, getpid());
    perror ("can't open client socket");
    exit (1);
  }

  /*** keep track of the highest socket number
  ***/
  if (remote_link[count].socket > max_sock_num)
       max_sock_num = remote_link[count].socket;

  /*** wait until associated remote port numbers are advertised
  ***/
  strcpy (key_string, "net/port/");
  strcat (key_string, remote_link[count].host_name);
  while (db_fetch (key_string, content_string) < 0)
  {  sleep (2);
  }
  sscanf (content_string, "%d", &remote_link[count].port);
  remote_link[count].port = ntohs (remote_link[count].port);
}

/*** Connect to all remote hosts whose local sockets were initialized
       above (in the same order).
***/
for (count = 1;
       count < HOST_NUMBER;
       ++count)
{  if ((host_ptr = gethostbyname (remote_link[count].host_name)) == 0)
```

```
  { printf ("%d: %d: '%s' unknown host in connect_to_remote...0,
              HOST_NUMBER, getpid(), remote_link[count].host_name);
  }
  bcopy (host_ptr -> h_addr, &server.sin_addr, host_ptr -> h_length);
  server.sin_family = AF_INET;
  server.sin_port = htons (remote_link[count].port);
  while (connect (remote_link[count].socket, &server,sizeof(server)) < 0){}
}


/*** advertise the local node address for use unrelated to this specific
      algorithm
***/
strcpy (key_string, "net/cntrl/");
strcat (key_string, HOST_NAME);
sprintf (content_string, "%d", local_node_addr);
cntrl_db_store (key_string, content_string);
sleep (1);


/*** Initialize a local socket to be used in accepting remote connection
      requests.
***/
if ((remote_link[HOST_NUMBER].socket =
      socket (AF_INET, SOCK_STREAM, 0)) < 0)
{ printf ("%d: %d: ", HOST_NUMBER, getpid());
  perror ("can't open server socket");
  exit (1);
}


/*** bind that socket to an Internet address
***/
server.sin_family = AF_INET;
server.sin_addr.s_addr = INADDR_ANY;
server.sin_port = 0;
if (bind (remote_link[HOST_NUMBER].socket, &server, sizeof (server)))
{ printf ("%d: %d: ", HOST_NUMBER, getpid());
  perror ("can't bind server socket to an internet address");
  exit (1);
}


/*** extract the port number from the Internet address
***/
length = sizeof (server);
```

```
  if (getsockname (remote_link[HOST_NUMBER].socket, &server, &length))
  {  printf ("%d: %d: ", HOST_NUMBER, getpid ());
     perror ("can't get port assigned to server socket");
     exit (1);
  }
  remote_link[HOST_NUMBER].port = ntohs (server.sin_port);


  /*** advertise the port number for the local network controller
  ***/
  strcpy (key_string, "net/port/");
  strcat (key_string, HOST_NAME);
  sprintf (content_string, "%d", htons (remote_link[HOST_NUMBER].port));
  cntrl_db_store (key_string, content_string);
  sleep (1);


  /*** instruct the socket to start listening for connection requests
  ***/
  listen (remote_link[HOST_NUMBER].socket, 5);


  /*** Accept connection requests from all hosts with network
         identification numbers higher than the local host's.
         Accept them in increasing order as that is the way they
         will arrive.
  ***/
  for (count = HOST_NUMBER + 1;
         count <= NUMBER_OF_HOSTS;
         ++count)
  {  length = sizeof (server);
     if ((remote_link[count].socket =
            accept (remote_link[HOST_NUMBER].socket, 0, 0)) < 0)
     {  printf ("%d: %d: ", HOST_NUMBER, getpid());
           perror ("accepting client connection");
        exit (1);
     }
     if (remote_link[count].socket > max_sock_num)
           max_sock_num = remote_link[count].socket;
  }
}


start_local_net_server ()
{
}
```

```
poll_remote_connections ()
{
  fd_set read_set;
  struct timeval time;
  int  count;
  long msg_id,
        dest_node_addr,
        long_msg_type,
        long_data_size;
  char data_buff[MAX_HARD_COPY_SIZE];
  static long remote_msg_ack = REMOTE_MSG_ACK;
  msg_buff msg;


  /*** Build the a set of the sockets that are to be ckecked
        for incoming messages.
  ***/
  FD_ZERO (&read_set);
  for (count = 1;
        count <= NUMBER_OF_HOSTS;
        ++count)
  {  if (count != HOST_NUMBER)
        (void)FD_SET (remote_link[count].socket, &read_set);
  }


  /*** Poll the sockets for incoming messages
  ***/
  time.tv_sec = 0;
  time.tv_usec = 0;
  if (select (max_sock_num + 1, &read_set, 0, 0, &time) > 0)
  {  for (count = 1;
          count <= NUMBER_OF_HOSTS;
          ++count)
    {  if (count != HOST_NUMBER)
       {  if (FD_ISSET (remote_link[count].socket, &read_set) > 0)
            {  if (read (remote_link[count].socket,
                         (char *)&msg_id, 4) != 4)
          {  printf ("%d: %d: error reading msg id from socket0,
                         HOST_NUMBER, getpid());
             perror ("   ");
             exit (1);
          }

             if (read (remote_link[count].socket,
                   (char *)&long_msg_type, 4) != 4)
```

```
{  printf ("%d: %d: error reading msg type from socket0,
                HOST_NUMBER, getpid());
  perror ("   ");
        exit (1);
}

   if (long_msg_type == REMOTE_MSG_ACK)
   {  remote_link[count].ack_msg_id = msg_id;
        break;
}

   if (write (remote_link[count].socket,
        (char *)&msg_id, 4) != 4)
{  printf ("%d: %d: error writing ack id to socket0,
                HOST_NUMBER, getpid ());
  perror ("   ");
  exit (1);
}

   if (write (remote_link[count].socket,
        (char *)&remote_msg_ack, 4) != 4)
{  printf ("%d: %d: error writing ack type to socket0,
                HOST_NUMBER, getpid());
  perror ("   ");
  exit (1);
}

   if (read (remote_link[count].socket,
        (char *)&long_data_size, 4) != 4)
{  printf ("%d: %d: error reading msg size from sock0,
                HOST_NUMBER, getpid());
  perror ("   ");
  exit (1);
}

   if (read (remote_link[count].socket,
        &data_buff[0], long_data_size) != long_data_size)
{  printf ("%d: %d: error reading msg data from sock0,
                HOST_NUMBER, getpid());
  perror ("   ");
  exit (1);
}


   bcopy (&data_buff[0], &dest_node_addr, 4);
make_msg (data_buff, (int)long_data_size,
```

```
                    ANY_MSG_TYPE, REFERENCE_ONLY, &msg);
            (void)net_send (dest_node_addr, &msg,
                    RETURN_IF_NO_ACKNOWLEDGE, DISREGARD_ACKNOWLEDG
        }
      }
    }
  }
}



/*** This procedure polls and accepts requests from any local node process.
***/
poll_local_connections ()
{
  msg_buff msg;

  if (net_poll (ALL_NODE_ADDR, REFERENCE_AND_COPY, &msg) == 0)
  {  switch (msg.msg_type)
     {  /*** Facilities to handle these first three types of requests
            are not yet implemented.
       case SUBNET_CREATE: subnet_create (&msg);
                   break;
       case SUBNET_DELETE: subnet_delete (&msg);
                       break;
       case NETWORK_SHUTDOWN: network_shutdown (&msg);
                        break;
       ***/


         /*** Facilities to handle these next four types of requests
            are implemented but not yet tested.
       case SHD_MEM_REQ_CONN: if (relay_conn_msg (&msg) == 0)
                          ack_msg (&msg);
                   break;
       case SHD_MEM_ACK_CONN: if (relay_conn_msg (&msg) == 0)
                         ack_msg (&msg);
                          break;
       case SHD_MEM_REQ_BREAK: (void)ack_break (&msg);
                          ack_msg (&msg);
                   break;
       case SHD_MEM_ACK_BREAK: (void)break_conn (&msg);
                          break;
       ***/
```

```
        case FORWARD_MSG: if (forward_remote_msg (&msg) == 0)
                                ack_msg (&msg);
                        else
                                nack_msg (&msg);
                        break;
        default: nack_msg (&msg);
                break;
      }
    }
}



update_remote_shd_mem ()
{
}



/*** Allow new local node processes to establish shared memory IPC
    with the network controller.
***/
service_net_connection_request ()
{
  static int last_request_pid,
            *request_pid_ptr,
              *service_pid_ptr,
              *acknowledge_ptr,
              *assigned_segment_offset_ptr,
              *assigned_link_offset_ptr,
              *cntrl_link_ptr,
              request_init_flag = -1;
  int link_assigned_flag = -1,
      count;


  /*** Perform initializations the first time this process is called.
  ***/
  if (request_init_flag == -1)
  {  request_pid_ptr = (int *)shd_mem_first_addr;
     *request_pid_ptr = 0;
     last_request_pid = 0;
     service_pid_ptr = (int *)shd_mem_first_addr + 1;
     *service_pid_ptr = 0;
     acknowledge_ptr = (int *)shd_mem_first_addr + 2;
     *acknowledge_ptr = 0;
```

```
    assigned_segment_offset_ptr = (int *)shd_mem_first_addr + 3;
    assigned_link_offset_ptr = (int *)shd_mem_first_addr + 4;
    cntrl_link_ptr = (int *)shd_mem_first_addr + 5;
    request_init_flag = 1;
    return (-1);
  }


  if (*acknowledge_ptr == -1)
  { /*** The last requesting process that was given connection
          informtion has not yet received it.
    ***/
    return (-1);
  }
  if (*acknowledge_ptr == 1)
  { /*** The last requesting process has received the connection
          information and has finished initializations so that the
          logical link may now go 'active'.
    ***/
    link[*cntrl_link_ptr].remote_node_addr =
        - link[*cntrl_link_ptr].remote_node_addr;
    if (*cntrl_link_ptr > high_link_id)
        high_link_id = *cntrl_link_ptr;
    *acknowledge_ptr = 0;
printf ("%d: %d: control link assigned: %d0, HOST_NUMBER, getpid(),
        *cntrl_link_ptr);
  }


  /*** If there is a new request present then handle it.
        Find the first available link and assign it to the
        requesting process. (use only the first five links)
  ***/
  if (*request_pid_ptr != last_request_pid)
  { last_request_pid = *request_pid_ptr;
    for (count = 0;
          (count < 5) && (link_assigned_flag == -1);
          ++count)
    { if (link[count].remote_node_addr == -1)
        { /*** A free link has been found for initialization but start it
                off in an 'inactive' state.
          ***/
          link[count].remote_node_addr =
            - ((HOST_NUMBER * 0x10000000) + (last_request_pid * 0x2000));
```

```
        /*** Assign the offset as the distance from the first address of
                the main shared memory segment that the logical segment of
                the requesting process is to be located.
        ***/
        link[count].remote_segment_offset =
             SHD_MEM_SPACE_SIZE  * (count + 1);
        /*** initialize standard shared memory link info
           ***/
        link[count].remote_msg_id_ptr = (long *)((long)shd_mem_first_addr +
             link[count].remote_segment_offset + SHD_MEM_LINK_SPACE_OFFSET);
        link[count].remote_msg_offset_ptr =
             (long *)(link[count].remote_msg_id_ptr + 1);
           link[count].remote_ack_id_ptr  = link[count].remote_msg_id_ptr + 2;
        *assigned_segment_offset_ptr = link[count].remote_segment_offset;
           *assigned_link_offset_ptr =
             (int)((long)link[count].local_msg_id_ptr -
                 (long)shd_mem_first_addr);
        *cntrl_link_ptr = count;
        *acknowledge_ptr = -1;
        *service_pid_ptr = last_request_pid;
           link_assigned_flag = 1;
      }
    }
    return (0);
  }
  else
    return (-1);
}


/*** Relay data between node processes that are connected to the
    network controller but not to each other.
***/
forward_remote_msg (msg_ptr)
  msg_buff *msg_ptr;
{
  long dest_node_addr,
      remote_host,
         long_msg_type,
         long_data_size;
  msg_buff *relay_ptr;

  bcopy (msg_ptr -> ref_data_ptr, &dest_node_addr, 4);
  if ((remote_host =
         (dest_node_addr & MSG_ID_TO_HOST_MASK) / 0x10000000)
         == HOST_NUMBER)
```

```
{printf ("%d: %d: host is local host0,
        HOST_NUMBER, getpid());
  return (-1);
}
if ((remote_host < 1) || (remote_host > NUMBER_OF_HOSTS))
{printf ("%d: %d: host out of range: %x  addr: %x0,
        HOST_NUMBER, getpid(), remote_host,
        (long)*msg_ptr -> ref_data_ptr);
  return (-1);
}
if (remote_link[remote_host].sent_msg_id !=
   remote_link[remote_host].ack_msg_id)
  return (-1);

if (write (remote_link[remote_host].socket,
   &msg_ptr -> hard_copy_id, 4) != 4)
{  printf ("%d: %d: writing msg id over socket ",
           HOST_NUMBER, getpid ());
  perror ();
  exit (1);
}

long_msg_type = (long)msg_ptr -> msg_type;
if (write (remote_link[remote_host].socket,
   &long_msg_type, 4) != 4)
{  printf ("%d: %d: writing msg type over socket ",
           HOST_NUMBER, getpid());
  perror ();
  exit (1);
}

long_data_size = (long)msg_ptr -> hard_copy_size;
if (write (remote_link[remote_host].socket,
   &long_data_size, 4) != 4)
{  printf ("%d: %d: writing msg data size over socket ",
           HOST_NUMBER, getpid());
  perror ();
  exit (1);
}

if (write (remote_link[remote_host].socket,
   msg_ptr -> hard_copy_data,
   msg_ptr -> hard_copy_size) != msg_ptr -> hard_copy_size)
{  printf ("%d: %d: writing msg data over socket ",
           HOST_NUMBER, getpid());
  perror ();
```

```
    exit (1);
  }
  remote_link[remote_host].sent_msg_id = msg_ptr -> hard_copy_id;
  return (0);
}



/*** This  procedure is to be used when network server facilities
    are not available. Note: it does not use any locking controls!
***/
cntrl_db_store (key_string, content_string)
  char key_string[],
         content_string[];
{
  char syscmd[128];

  strcpy (syscmd, "/work/frank/network/dbcntrl net s g ");
  strcat (syscmd, key_string);
  strcat (syscmd, " ");
  strcat (syscmd, content_string);
  strcat (syscmd, " &");
  system (syscmd);
}
```

```
/*************************************************************************

    This is a UNIX V module called netipc.c. It is part of thesis work
    done summer and fall quarter 1987 by Lt. W. Cory Frank under the
    title, "UNIX Based Programming Tools for Locally Distributed Network
    Applications. It's main function is to provide programmers with the
    benefit of shared memory IPC on a higher level of abstraction than
    the canned UNIX V tools. Together with netcntrl.c,  it provides a
    transparent link to remote IPC over an experimental logical network
    on Ethernet.

*************************************************************************/

#include "netglobal.h"
#include "netlocal.h"
#include "netipc.h"
#include "netmsgtypes.h"
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>

/*** data structure for a processes logical links to other processes
    via shared memory
***/
shd_mem_link_info link[SHD_MEM_MAX_LINKS];

/*** this is always one less than the next message number to be sent
***/
static long msg_id_num = -1;

/*** these are specially formatted network addresses for the network
    controller process and local process respectively
***/
long local_node_addr,
    cntrl_node_addr;


int  cntrl_shd_mem_id,          /*** system assigned shared memory id ***/
    cntrl_segment_offset = 0,  /*** control process uses start of memory ***/
    local_segment_offset,      /*** local process offset from start of mem */
    high_link_id = 0,          /*** must keep track of highest link in use */
    shd_mem_init_flag = -1;    /*** keep track of init status of shared mem*/
char *shd_mem_first_addr,       /*** system assigned value ***/
    *local_segment_first_addr,
    *link_segment_first_addr,  /*** shared mem for local process IPC links */
```

```
    *link_segment_last_addr,
    *msg_segment_first_addr,   /*** shared mem for local process messages **/
    *msg_segment_last_addr,
    *adt_segment_first_addr,   /*** shared mem for local process to use for */
    *adt_segment_last_addr;    /*** data meant for all to read **/


/*** This procedure provides a high level of abstraction for making
    a message data structure suitable for shared memory IPC
***/
make_msg (data_buff, data_size,
                    msg_type, msg_mode, msg_ptr)
  char data_buff[];
  int  data_size,
         msg_type,
       msg_mode;
  msg_buff *msg_ptr;
{
  /*** keep track of spot to place next message in the local process
         shared memory message space
  ***/
  static char *ring_buffer_ptr;

  /*** allow the ring_buffer_ptr to be set to the first byte of message
         space the first time called
  ***/
  static int  ring_buffer_init_flag = -1;

  long long_data_size,   /*** always use long integers (4-bytes) ***/
         long_msg_type,    /*** in shared memory; fewer bytes cause trouble ***/
       msg_bytes;
  int  rounded_data_size;

  if (ring_buffer_init_flag == -1)
  {  ring_buffer_ptr = msg_segment_first_addr;
     ring_buffer_init_flag = 1;
  }
  switch (msg_mode)
  {  case REFERENCE_ONLY: break;
     case REFERENCE_AND_COPY: break;
     default: printf("%d: %d: unrecognized msg_mode in make_msg.0,
                        HOST_NUMBER, getpid());
              exit (1);
  }
```

```
/*** message size must be a 4-byte multiple
***/
rounded_data_size = (data_size / 4) * 4;

msg_bytes = SHD_MEM_MSG_NON_DATA_BYTES + rounded_data_size;
if (msg_bytes > SHD_MEM_MSG_SPACE_SIZE)
{  printf ("%d: %d: data_size is too large in make_msg.0,
           HOST_NUMBER, getpid());
  exit (1);
}

/*** if necessary, rotate to the beginning of the message space before
       the new message is placed there
***/
if (((long)ring_buffer_ptr + msg_bytes) >
   (long)msg_segment_last_addr)
  ring_buffer_ptr = msg_segment_first_addr;

/*** increment to next message id number before assigning
       but start with 0 again if the range has been exeeded
***/
msg_id_num = (msg_id_num + 1) % (MAX_MSG_ID + 1);

/*** build and assign the network message id
***/
msg_ptr -> hard_copy_id = msg_id_num | local_node_addr;

/*** initialized for later assignment
***/
msg_ptr -> sent_link = -1;
msg_ptr -> rcvd_link = -1;

/*** assign and place all message parameters in ring buffer
***/
msg_ptr -> ref_msg_ptr = ring_buffer_ptr;
msg_ptr -> msg_mode = msg_mode;
msg_ptr -> ref_id_ptr = (long *)((long)ring_buffer_ptr +
                          SHD_MEM_MSG_ID_OFFSET);
*msg_ptr -> ref_id_ptr = msg_ptr -> hard_copy_id;
msg_ptr -> msg_type = msg_type;
long_msg_type = (long)msg_type;
bcopy (&long_msg_type,
        ring_buffer_ptr + SHD_MEM_MSG_TYPE_OFFSET, 4);
msg_ptr -> ref_data_size = data_size;
long_data_size = (long)data_size;
bcopy (&long_data_size, ring_buffer_ptr + SHD_MEM_MSG_DATA_SIZE_OFFSET, 4);
```

```
  msg_ptr -> ref_data_ptr = ring_buffer_ptr + SHD_MEM_MSG_DATA_OFFSET;
  bcopy (data_buff, msg_ptr -> ref_data_ptr, rounded_data_size);
  if (msg_mode == REFERENCE_AND_COPY)
  {  if (data_size > MAX_HARD_COPY_SIZE)
         msg_ptr -> hard_copy_size = MAX_HARD_COPY_SIZE;
     else
         msg_ptr -> hard_copy_size = data_size;
     bcopy (data_buff, msg_ptr -> hard_copy_data,
             msg_ptr -> hard_copy_size);
  }
  else
     msg_ptr -> hard_copy_size = 0;

  ring_buffer_ptr = (char *)((long)ring_buffer_ptr + msg_bytes);
  return (0);
}

/*** Allow a single shared memory IPC link to be checked for an
    incoming message. If there is one then read it; if not then return.
***/
shd_mem_poll_link (link_id, msg_mode, msg_ptr)
  int link_id,
      msg_mode;
  msg_buff *msg_ptr;
{
  long long_msg_type,
        long_data_size;

  if ((link_id < 0) || (link_id > SHD_MEM_MAX_LINKS - 1))
  {  printf ("%d: %d: link_id out of range in shd_mem_poll_link.0,
             HOST_NUMBER, getpid());
     exit (1);
  }
  if (link[link_id].remote_node_addr < 0)
  {  printf ("%d: %d: attempt to poll inactive link0,
             HOST_NUMBER, getpid());
     exit (1);
  }

  switch (msg_mode)
  { case REFERENCE_ONLY: break;
    case REFERENCE_AND_COPY: break;
    default: printf ("%d: %d: unrecognized msg_mode in poll_link.0,
                     HOST_NUMBER, getpid());
         exit (1);
  }
```

```
  /*** If the id of the message has not been either positively or
       negatively acknowledged yet by the local process then assume
       it is new.
  ***/
  if ((*link[link_id].remote_msg_id_ptr !=
     *link[link_id].local_ack_id_ptr) &&
    (*link[link_id].remote_msg_id_ptr !=
       *link[link_id].local_ack_id_ptr))
  {  msg_ptr -> rcvd_link = link_id;
    msg_ptr -> msg_mode = msg_mode;
    msg_ptr -> hard_copy_id = *link[link_id].remote_msg_id_ptr;
    msg_ptr -> ref_msg_ptr = (char *)((long)shd_mem_first_addr +
         link[link_id].remote_segment_offset +
         *link[link_id].remote_msg_offset_ptr);
    bcopy (msg_ptr -> ref_msg_ptr + SHD_MEM_MSG_TYPE_OFFSET,
           &long_msg_type, 4);
    msg_ptr -> msg_type = (int)long_msg_type;
    msg_ptr -> ref_id_ptr = (long *)(msg_ptr -> ref_msg_ptr +
                              SHD_MEM_MSG_ID_OFFSET);
    msg_ptr -> ref_data_ptr = msg_ptr -> ref_msg_ptr +
                                 SHD_MEM_MSG_DATA_OFFSET;
    bcopy (msg_ptr -> ref_msg_ptr + SHD_MEM_MSG_DATA_SIZE_OFFSET,
           &long_data_size, 4);
    msg_ptr -> ref_data_size = (int)long_data_size;
    if (msg_mode == REFERENCE_AND_COPY)
    {  if (msg_ptr -> ref_data_size < MAX_HARD_COPY_SIZE)
           msg_ptr -> hard_copy_size = msg_ptr -> ref_data_size;
      else
           msg_ptr -> hard_copy_size = MAX_HARD_COPY_SIZE;
      bcopy (msg_ptr -> ref_data_ptr, msg_ptr -> hard_copy_data,
                msg_ptr -> hard_copy_size);
    }
    if (*msg_ptr -> ref_id_ptr != msg_ptr -> hard_copy_id)
         return (-1);
    else
      return (0);
  }
  else
    return (-1);
}


/*** Allow a single or all links to be polled for an incoming message.
***/
net_poll (remote_node_addr, msg_mode, msg_ptr)
```

```
  long remote_node_addr;
  int  msg_mode;
  msg_buff *msg_ptr;
{
  static int next_link_id = 0;
  int count,
     link_found_flag = -1;

  if (remote_node_addr == ALL_NODE_ADDR)
  {  for (count = 0;
            count <= high_link_id;
            ++count)
     {  if (link[next_link_id].remote_node_addr >= 0)
        {  if (shd_mem_poll_link (next_link_id, msg_mode, msg_ptr) == 0)
              {  next_link_id = (next_link_id + 1) % (high_link_id + 1);
            return (0);
          }
        }
           next_link_id = (next_link_id + 1) % (high_link_id + 1);
     }
     return (-1);
  }
  else
  {  for (count = -1;
         (count <= high_link_id) && (link_found_flag == -1);
            count++)
     {  if (link[count].remote_node_addr == remote_node_addr)
              link_found_flag = 1;
     }
     if (link_found_flag == -1)
     {  printf ("%d: %d: attempt to poll unconnected node0,
                    HOST_NUMBER, getpid());
        exit (1);
     }
     else
     {  if (shd_mem_poll_link (count, msg_mode, msg_ptr) == 0)
           return (0);
        else
              return (-1);
     }
  }
}


/*** Allow blocking on a single or all links until a message is received.
```

```
***/
net_recv (remote_node_addr, msg_mode, msg_ptr)
  long remote_node_addr;
  int  msg_mode;
  msg_buff *msg_ptr;
{
  while (net_poll (remote_node_addr, msg_mode, msg_ptr) == -1){}
  return (0);
}


/*** Allow blocking on a single link until a message is received.
***/
shd_mem_recv_link (link_id, msg_mode, msg_ptr)
  int link_id,
      msg_mode;
  msg_buff *msg_ptr;
{
  while (shd_mem_poll_link (link_id, msg_mode, msg_ptr) == -1){}
  return (0);
}


/*** Send a properly formatted message to another process (node).
***/
net_send (dest_node_addr, msg_ptr, pre_send_protocol, post_send_protocol)
  long dest_node_addr;
  int  pre_send_protocol,
       post_send_protocol;
  msg_buff *msg_ptr;
{
  int count,
      found_link = -1;

  for (count = 0;
         (count <= SHD_MEM_MAX_LINKS - 1) && (found_link == -1);
         ++count)
  {  if (link[count].remote_node_addr == dest_node_addr)
          found_link = count;
  }
  if (found_link == -1)
  { printf ("%d: %d: attempt to send to unconnected node0,
             HOST_NUMBER, getpid());
    exit (1);
```

```
  }
  return (shd_mem_send_link (found_link, msg_ptr,
                             pre_send_protocol, post_send_protocol));
}



/*** Send a message over a selected shared memory link.
***/
shd_mem_send_link (link_id, msg_ptr, pre_send_protocol, post_send_protocol)
  int link_id,
     pre_send_protocol,
     post_send_protocol;
  msg_buff *msg_ptr;
{
  if ((link_id < 0) || (link_id > SHD_MEM_MAX_LINKS - 1))
  {  printf ("%d: %d: link_id out of range in send_link.0,
             HOST_NUMBER, getpid());
    exit (1);
  }
  if (link[link_id].remote_node_addr < 0)
  {  printf ("%d: %d: attempt to use inactive link in send_link0,
             HOST_NUMBER, getpid());
    exit (1);
  }

  switch (pre_send_protocol)
  {  case RETURN_IF_NO_ACKNOWLEDGE: break;
    case WAIT_FOR_ACKNOWLEDGE: break;
    case DISREGARD_ACKNOWLEDGE: break;
    default: printf ("%d: %d: unrecognized pre_send_protocol0,
                     HOST_NUMBER, getpid());
             exit (1);
  }
  switch (post_send_protocol)
  {  case WAIT_FOR_ACKNOWLEDGE: break;
    case DISREGARD_ACKNOWLEDGE: break;
    default: printf ("%d: %d: unrecognized post_send_protocol0,
                     HOST_NUMBER, getpid());
             exit (1);
  }

  if (*(msg_ptr -> ref_id_ptr) != msg_ptr -> hard_copy_id)
  {  printf ("   refidptr: %d    contents: %x0,
             msg_ptr -> ref_id_ptr, *msg_ptr -> ref_id_ptr);
    return (-1);
```

```
  }

  switch (pre_send_protocol)
  {  case RETURN_IF_NO_ACKNOWLEDGE:
          if ((*link[link_id].remote_ack_id_ptr !=
              *link[link_id].local_msg_id_ptr) &&
          (*link[link_id].remote_ack_id_ptr !=
              - *link[link_id].local_msg_id_ptr))
         return (-1);
       break;
     case WAIT_FOR_ACKNOWLEDGE:
       while ((*link[link_id].remote_ack_id_ptr !=
             *link[link_id].local_msg_id_ptr) &&
            (*link[link_id].remote_ack_id_ptr !=
                  - *link[link_id].local_msg_id_ptr)){}
       break;
     case DISREGARD_ACKNOWLEDGE: break;
  }

  /*** Advertise the info about the new message for the process on the
         other end of the link.
  ***/
  *link[link_id].local_msg_offset_ptr =
    ((long)msg_ptr -> ref_msg_ptr) - ((long)local_segment_first_addr);
  *link[link_id].local_msg_id_ptr = msg_ptr -> hard_copy_id;

  if (post_send_protocol == WAIT_FOR_ACKNOWLEDGE)
  {  while ((*link[link_id].remote_ack_id_ptr !=
              *link[link_id].local_msg_id_ptr) &&
          (*link[link_id].remote_ack_id_ptr !=
              - *link[link_id].local_msg_id_ptr)){}
    if (*link[link_id].remote_ack_id_ptr < 0)
          return (-1);
  }
  msg_ptr -> sent_link = link_id;
  return (0);
}



/*** Relay a properly formatted message through the network controller
    process when establishing a shared memory link with another process.
***/
relay_msg (msg_ptr)
  msg_buff *msg_ptr;
{
```

```
  long dest_addr,
        return_addr;
  msg_buff relay_msg;

  bcopy (msg_ptr -> hard_copy_data, &dest_addr, 4);
  if (dest_addr != local_node_addr)
  {  return_addr = msg_ptr -> hard_copy_id | MSG_ID_TO_ADDR_MASK;
     bcopy (&return_addr, msg_ptr -> hard_copy_data, 4);
     make_msg (msg_ptr -> hard_copy_data, 8,
                    SHD_MEM_REQ_CONN, REFERENCE_ONLY, &relay_msg);
     if (net_send (dest_addr, &relay_msg,
                     RETURN_IF_NO_ACKNOWLEDGE, DISREGARD_ACKNOWLEDGE) < 0)
       return (-1);
     else
          return (0);
  }
  else
     return (-1);
}



/*** Positively acknowledge a message.
***/
ack_msg (msg_ptr)
  msg_buff *msg_ptr;
{
  *link[msg_ptr -> rcvd_link].local_ack_id_ptr =
     (msg_ptr -> hard_copy_id);
  return (0);
}



/*** Negatively acknowledge a message.
***/
nack_msg (msg_ptr)
  msg_buff *msg_ptr;
{
  *link[msg_ptr -> rcvd_link].local_ack_id_ptr =
     - (msg_ptr -> hard_copy_id);
  return (0);
}
```

```
/*** Initialize shared memory for local process use before attempting
    any other shared memory IPC operations.
***/
shd_mem_init ()
{
  int  count,
         local_pid,
         *request_pid_ptr, /*** buffer for any requesting process to use ***/
         *service_pid_ptr, /*** shows requester being serviced by cntrl node **/
         *acknowledge_ptr, /*** used by local process to advertise when done **/
         *assigned_segment_offset_ptr,/*** cntrl node gives shd mem instr here*/
         *cntrl_link_offset_ptr, /*** cntrl node gives link info here ***/
         cntrl_link_offset;   /*** shows where the control node holds its **/
                              /*** share of the link info for the local node */
  char logical_addr_string[32],
         cntrl_node_string[32];

  if (shd_mem_init_flag == 1)
    return (0);

  /*** Establish the local node's network address.
  ***/
  local_node_addr = (HOST_NUMBER * 0x10000000) + (getpid() * 0x2000);

  /*** Make the key necessary for retreiving the address of the control
         node from the global database.
  ***/
  strcpy (logical_addr_string, "net/cntrl/");
  strcat (logical_addr_string, HOST_NAME);

  /*** Get the control node's address.
  ***/
  if (db_fetch (logical_addr_string, cntrl_node_string) < 0)
  {  printf ("%d: %d: control addr not available in shd_mem_init0,
             HOST_NUMBER, getpid());
    exit (1);
  }
  sscanf (cntrl_node_string, "%d", &cntrl_node_addr);

  /*** If the local process is the control process then create the shared
         memory segment to be used for IPC.
  ***/
  if (local_node_addr == cntrl_node_addr)
  {  local_segment_offset = 0;
    if ((cntrl_shd_mem_id = shmget ((key_t)cntrl_node_addr,
                                    SHD_MEM_NUM_SEGMENTS * SHD_MEM_SPACE_SI
```

```
                                        0666 | IPC_CREAT)) == -1)
  { printf ("%d: %d ", HOST_NUMBER, getpid());
    perror ("shmget 'create' failure in shd_mem_init");
      exit (1);
  }
}
else
/*** Get the key shared memory id before attaching to it.
***/
{ if ((cntrl_shd_mem_id = shmget ((key_t)cntrl_node_addr,
                                        SHD_MEM_NUM_SEGMENTS * SHD_MEM_SPACE_SIZE,
                                        0666)) == -1)
  { printf ("%d: %d: ", HOST_NUMBER, getpid());
      perror ("shmget 'get cntrl id' failure in shd_mem_init");
      exit (1);
  }
}

/*** Attach to the entire shared memory segment.
***/
if ((shd_mem_first_addr =
   (char *)shmat (cntrl_shd_mem_id, (char *)0, 0)) == (char *)(-1))
{ printf ("%d: %d: ", HOST_NUMBER, getpid());
  perror ("Attaching to cntrl shared memory segment in shd_mem_init");
  exit (1);
}

if (local_node_addr != cntrl_node_addr)
/*** Get assigned shared memory and link info from control node.
***/
{ /*** initialize pointers to buffers in the control node's
        shared memory space.
  ***/
  request_pid_ptr = (int *)shd_mem_first_addr;
  service_pid_ptr = (int *)shd_mem_first_addr + 1;
  acknowledge_ptr = (int *)shd_mem_first_addr + 2;
  assigned_segment_offset_ptr = (int *)shd_mem_first_addr + 3;
  cntrl_link_offset_ptr = (int *)shd_mem_first_addr + 4;

  local_pid = getpid ();
  /*** Keep trying for service until it is granted.
  ***/
  while (*service_pid_ptr != local_pid)
  { if (*request_pid_ptr != local_pid)
      *request_pid_ptr = local_pid;
  }
```

```
  /*** Pick up assigned shared memory and link info.
  ***/
  local_segment_offset = *assigned_segment_offset_ptr;
  cntrl_link_offset = *cntrl_link_offset_ptr;

  /*** Initialize link to control node.
  ***/
  link[0].remote_node_addr = cntrl_node_addr;
  link[0].remote_segment_offset = 0;
  link[0].remote_msg_id_ptr = (long *)((long)shd_mem_first_addr +
                                        cntrl_link_offset);
  link[0].remote_msg_offset_ptr =
       (long *)(link[0].remote_msg_id_ptr + 1);
  link[0].remote_ack_id_ptr  = link[0].remote_msg_id_ptr + 2;
}

/*** Initialize local shared memory info.
***/
local_segment_first_addr = shd_mem_first_addr +
                           local_segment_offset;
adt_segment_first_addr = local_segment_first_addr +
                         SHD_MEM_ADT_SPACE_OFFSET;
adt_segment_last_addr  = adt_segment_first_addr +
                         SHD_MEM_ADT_SPACE_SIZE - 1;
msg_segment_first_addr = local_segment_first_addr +
                         SHD_MEM_MSG_SPACE_OFFSET;
msg_segment_last_addr  = msg_segment_first_addr +
                         SHD_MEM_MSG_SPACE_SIZE - 1;
link_segment_first_addr = local_segment_first_addr +
                          SHD_MEM_LINK_SPACE_OFFSET;
link_segment_last_addr  = link_segment_first_addr +
                          SHD_MEM_LINK_SPACE_SIZE - 1;

/*** Finish initializing all local link info.
***/
for ( count = 0;
      count <= SHD_MEM_MAX_LINKS - 1;
      ++count)
{  if (local_node_addr == cntrl_node_addr)
     link[count].remote_node_addr = -1;
  link[count].local_msg_id_ptr = (long *)link_segment_first_addr +
                                 (count * 3);
  *link[count].local_msg_id_ptr = -1;
  link[count].local_msg_offset_ptr =
        (long *)(link[count].local_msg_id_ptr + 1);
  link[count].local_ack_id_ptr  = link[count].local_msg_id_ptr + 2;
```

```
    *link[count].local_ack_id_ptr = -1;
  }

  /*** Tell the control node that you are finished initializing and active.
  ***/
  if (local_node_addr != cntrl_node_addr)
    *acknowledge_ptr = 1;

  shd_mem_init_flag = 1;
  return (0);
}



/*** Detach from the shared memory segment.
***/
shd_mem_detach ()
{
  struct shmid_ds *buf;

  shmdt (shd_mem_first_addr);
  if (local_node_addr == cntrl_node_addr)
    shmctl (cntrl_shd_mem_id, IPC_RMID, buf);
}



/*** Used to initiate a link to another process in the logical network.
***/
req_conn (remote_node_addr)
  long remote_node_addr;
{
  int  assignment_link = -1,
         duplicate_link = -1,
       count;
  long offset;
  char data_buff [8];
  msg_buff msg;

  for (count = -1;
         count <= SHD_MEM_MAX_LINKS - 1;
         count++)
  {  if ((link[count].remote_node_addr == -1) && (assignment_link == -1))
     { link[count].remote_node_addr = - remote_node_addr;
         assignment_link = count;
     }
```

```
    if (link[count].remote_node_addr == remote_node_addr)
          duplicate_link = count;
  }
  if ((assignment_link == -1) || (duplicate_link != -1))
    return (-1);
  /*** The control node must be told the address of the node you wish
          to connect to.
  ***/
  bcopy (remote_node_addr, data_buff, 4);
  offset = (long)((long)link[assignment_link].local_msg_id_ptr -
                  (long)shd_mem_first_addr);
  bcopy (&offset, data_buff + 4, 4);
  make_msg (data_buff, 8,
            SHD_MEM_REQ_CONN, REFERENCE_ONLY, &msg);
  if (shd_mem_send_link (SHD_MEM_CNTRL_LINK, &msg,
      RETURN_IF_NO_ACKNOWLEDGE, DISREGARD_ACKNOWLEDGE) < 0)
    return (-1);
  else
    return (assignment_link);
}


/*** Used to respond to a connection request.
***/
ack_conn (msg_ptr)
  msg_buff *msg_ptr;
{
  int count,
     assignment_link = -1,
     duplicate_link = -1,
     conn_denied_flag = -1;
  long remote_node_addr,
        offset;
  char data_buff[8];
  msg_buff reply_msg;

  bcopy (msg_ptr -> hard_copy_data, &remote_node_addr, 4);
  bcopy (msg_ptr -> hard_copy_data, data_buff, 4);

  for (count = 0;
      count <= SHD_MEM_MAX_LINKS -1;
      count++)
  {  if ((link[count].remote_node_addr == -1) && (assignment_link == -1))
       assignment_link = count;
    if (link[count].remote_node_addr == remote_node_addr)
          duplicate_link = count;
```

```
  }
  if ((assignment_link == -1) || (duplicate_link != -1))
  {  bcopy (&conn_denied_flag, data_buff + 4, 4);
  }
  else
  {  link[assignment_link].remote_node_addr = remote_node_addr;
     bcopy (msg_ptr -> hard_copy_data + 4, &offset, 4);
     link[assignment_link].remote_msg_id_ptr =
          (long *)((long)shd_mem_first_addr + offset);
     link[assignment_link].remote_msg_offset_ptr =
       (long *)((long)link[assignment_link].remote_msg_id_ptr + 4);
     link[assignment_link].remote_ack_id_ptr =
       link[assignment_link].remote_msg_id_ptr + 2;
     offset = (long)((char *)link[assignment_link].local_msg_id_ptr -
       shd_mem_first_addr);
     bcopy (&offset, data_buff + 4, 4);
     if (assignment_link > high_link_id)
       high_link_id = assignment_link;
  }
  make_msg (data_buff, 8,
          SHD_MEM_ACK_CONN, REFERENCE_ONLY, &reply_msg);
  if (assignment_link != -1)
  {  if (shd_mem_send_link (assignment_link, msg_ptr,
          DISREGARD_ACKNOWLEDGE, DISREGARD_ACKNOWLEDGE) < 0)
       return (-1);
     else
       return (assignment_link);
  }
  else
    return (duplicate_link);
}


/*** Used to complete a link when a response to a local request is received.
***/
complete_conn (msg_ptr)
  msg_buff *msg_ptr;
{
  int  count,
       assignment_link = -1;
  long remote_node_addr,
          offset;

  bcopy (msg_ptr -> hard_copy_data, &remote_node_addr, 4);

  for (count = 0;
```

```
      count <= SHD_MEM_MAX_LINKS -1;
      count++)
  {  if ( - link[count].remote_node_addr == remote_node_addr)
       assignment_link = count;
  }
  if (assignment_link == -1)
    return (-1);
  bcopy (msg_ptr -> hard_copy_data + 4, &offset, 4);
  link[assignment_link].remote_msg_id_ptr =
    (long *)((long)shd_mem_first_addr + offset);
  if (*link[assignment_link].remote_msg_id_ptr < 0)
  {  link[assignment_link].remote_node_addr = -1;
    return (-1);
  }
  else
  {  link[assignment_link].remote_node_addr =
          - link[assignment_link].remote_node_addr;
    if (assignment_link > high_link_id)
          high_link_id = assignment_link;
    return (assignment_link);
  }
}



/*** Used to initiate a coordinated termination of a commms with a node.
***/
req_break (remote_node_addr)
  long remote_node_addr;
{
  int count,
      found_flag = -1;

  for (count = 0;
         (count <= SHD_MEM_MAX_LINKS - 1) && (found_flag == -1);
         count++)
  {  if (link[count].remote_node_addr == remote_node_addr)
          found_flag = 1;
  }
  if (found_flag == -1)
    return (-1);
  return (req_break_link (count));
}
```

```
/*** Used to initiate the shutdown of a specific link.
***/
req_break_link (link_id)
  int link_id;
{
  msg_buff msg;
  char dummy_data[2];
  int  dummy_size = 0;

  if (link[link_id].remote_node_addr == -1)
    return (0);

  make_msg (dummy_data, dummy_size,
                SHD_MEM_REQ_BREAK, REFERENCE_ONLY, &msg);
  if (shd_mem_send_link (link_id, &msg,
     DISREGARD_ACKNOWLEDGE, DISREGARD_ACKNOWLEDGE) < 0)
    return (-1);
  else
    return (0);
}


/*** Used to acknowledge a request to break comms with a node.
***/
ack_break (msg_ptr)
  msg_buff *msg_ptr;
{
  int count,
      found_flag = -1,
      dummy_size = 0;
  char dummy_data[2];
  long remote_node_addr;
  msg_buff reply_msg;

  remote_node_addr = msg_ptr -> hard_copy_id | MSG_ID_TO_ADDR_MASK;

  for (count = 0;
        (count <= SHD_MEM_MAX_LINKS - 1) && (found_flag == -1);
        count++)
  {  if (link[count].remote_node_addr == remote_node_addr)
       found_flag = 1;
  }
  if (found_flag == -1)
    return (-1);
  break_link (count);
  make_msg (dummy_data, dummy_size,
```

```
                    SHD_MEM_ACK_BREAK, REFERENCE_ONLY, &reply_msg);
  if (net_send (remote_node_addr, &reply_msg,
                DISREGARD_ACKNOWLEDGE, WAIT_FOR_ACKNOWLEDGE) < 0)
  {  printf ("%d: %d: failure of shd mem send from ack_break0,
            HOST_NUMBER, getpid());
  }
  return (0);
}



/*** Used to finish breaking comms when a response to a local request
    is received.
***/
break_conn (msg_ptr)
  msg_buff *msg_ptr;
{
  int count,
     found_link = -1;
  long remote_node_addr;

  remote_node_addr = msg_ptr -> hard_copy_id | MSG_ID_TO_ADDR_MASK;

  for (count = 0;
        (count <= SHD_MEM_MAX_LINKS - 1) && (found_link == -1);
        ++count)
  {  if (link[count].remote_node_addr == remote_node_addr)
         found_link = count;
  }
  if (found_link == -1)
    return (-1);
  return (break_link (found_link));
}



/*** Used to finish closing a specific link.
***/
break_link (link_id)
  int link_id;
{
  if ((link_id < 0) || (link_id > SHD_MEM_MAX_LINKS - 1))
  {  printf ("%x: link id out of range in break_link.0,
            local_node_addr);
    exit (1);
  }
```

```
  link[link_id].remote_node_addr = -1;
  *link[link_id].local_msg_id_ptr = -1;
  *link[link_id].local_ack_id_ptr = -1;
  if (link_id = high_link_id)
    for ( ; link[high_link_id].remote_node_addr == -1;
            high_link_id-- ){}
  return (0);
}
```

```
/*********************************************************************

   This is program dbcntrl.c which provides for global database
   operations using the UNIX 'dbm' facilities and remote system
   shell call. It is part of thesis work done by Lt. W. Cory Frank
   during summer and fall quarter 1987 under the title, "UNIX Based
   Programming Tools for Locally Distributed Network Applications."

*********************************************************************/

#include "netdb.h"
#include "netglobal.h"
#include "netlocal.h"

main (argc, argv)
  int argc;
  char *argv[];
{
  char database_name[MAX_NAME_LENGTH],
      database_path[MAX_PATH_LENGTH],
      command[2],
      scope[2],
         content_string[MAX_CONTENT_LENGTH];
  datum key,
       content;


  /*** Check for proper command line format.
  ***/
  check_format (argc, argv);

  assign_vars (argc, argv, database_name, database_path,
                 command, scope, &key, &content, content_string);

  /*** Initialize the existing database unless the request is to 're-create'.
  ***/
  if (command[0] != DBM_CREATE)
    if (dbminit (database_path) < 0)
     { printf ("%d: %d: can't initialize database ", HOST_NUMBER, getpid());
         perror ();
         exit (1);
     }

  execute_command (database_name, database_path,
                    command, scope, key, content, content_string);
}
```

```
check_format (argc, argv)
/*** check command line for proper format ***/
  int argc;
  char *argv[];
{
  if (argc < 4)
  {  give_format ();
     exit (1);
  }
}


/*** Command line usage instructions.
***/
give_format ()
{
  printf ("Command line format is: second arg-  database name0);
  printf ("                        third  arg-  command0);
  printf ("                        fourth arg-  scope0);
  printf ("                        fifth  arg-  key0);
  printf ("                        more  args-  content strings (opt)0);
}


/*** set variables from command line info
***/
assign_vars (argc, argv, database_name, database_path,
          command, scope, key_ptr, content_ptr, content_string)
  int  argc;
  char *argv[],
      database_name[],
      database_path[],
      command[],
      scope[],
         content_string[];
  datum *key_ptr,
       *content_ptr;
{
  short count;

  strcpy (database_name, argv[1]);

  strcpy (database_path, DATABASE_ROOT_PATH);
  strcat (database_path, database_name);

  strncpy (command, argv[2], 1);
```

```
  strncpy (scope, argv[3], 1);

  if (argc > 4)
  {  key_ptr -> dptr = argv[4];
     key_ptr -> dsize = strlen (argv[4]) + 1;
  }
  else
  {  key_ptr -> dptr = "dummy_key";
     key_ptr -> dsize = strlen ("dummy_key") + 1;
  }
  if (argc > 5)
  {  for (count = 5; count < argc; ++count)
     {  strcat (content_string, argv[count]);
           strcat (content_string, " ");
     }
     content_ptr -> dptr = content_string;
     content_ptr -> dsize = strlen (content_string) + 1;
  }
  else
  {  content_ptr -> dptr = "dummy_content";
     content_ptr -> dsize = strlen ("dummy_content") + 1;
  }
}



execute_command (database_name, database_path,
                 command, scope, key, in_content, in_content_string)
  char database_name[],
       database_path[],
       command[],
       scope[],
          in_content_string[];
  datum key,
        in_content;
{
  char out_content_string[MAX_CONTENT_LENGTH],
          directory_path[MAX_PATH_LENGTH],
          page_path[MAX_PATH_LENGTH];
  datum out_key,
          out_content;

  /*** If a global request, then start simmilar remote processes
          before executing the command locally. Note: exception is
          when the entire database is to be 're-created'.
  ***/
```

```
if ((scope[0] == GLOBAL) && (command[0] != DBM_CREATE))
  global_execute (database_name, command, key, in_content);


switch (command[0])
{  case DBM_CREATE:
       strcpy (directory_path, database_path);
       strcat (directory_path, ".dir");
       if (creat (directory_path, 0644) < 0)
       {  printf ("%s: can't create database directory file.0, HOST_NAME);
          exit (1);
    }
       strcpy (page_path, database_path);
       strcat (page_path, ".pag");
       if (creat (page_path, 0644) < 0)
       {  printf ("%s: can't create database page file.0, HOST_NAME);
          exit (1);
    }
       if (dbminit (database_path) < 0)
       {  printf ("%d: %d: can't initialize database ",HOST_NUMBER,getpid());
          perror ();
          exit (1);
    }
       out_key.dptr = "net/remotes";
       out_key.dsize = strlen ("net/remotes") + 1;
       out_content.dptr = REMOTE_NAMES;
       out_content.dsize = strlen (REMOTE_NAMES) + 1;
       if (store (out_key, out_content) < 0)
       {  printf ("%d: %d: can't store net/remotes ", HOST_NUMBER, getpid());
          perror ();
          exit (1);
    }
       if (scope[0] == GLOBAL)
          global_execute (database_name, command, key, in_content);
    break;
  case DBM_STORE:
       if (store (key, in_content) < 0)
       {  printf ("%d: %d: can't store string ", HOST_NUMBER, getpid());
      perror ();
          exit (1);
    }
       break;
  case DBM_FETCH:
       out_content = fetch (key);
       if (out_content.dptr == 0)
       {  printf ("%s: error retrieving content by key.0, HOST_NAME);
```

```
            exit (1);
        }
          else
          {  strncpy (out_content_string, out_content.dptr, out_content.dsize);
             printf ("%s:  %s -> %s0, HOST_NAME, key.dptr, out_content_string);
          }
          break;
      case DBM_DELETE:
          if (delete (key) < 0)
          {  printf ("%s: error deleting key/content pair.0, HOST_NAME);
             exit (1);
        }
          break;
    }
}


/*** Use remote system shell calls to duplicate the command on other
    hosts within the logical network.
***/
global_execute (database_name, command, key, content)
  char database_name[],
       command[];
  datum key,
          content;
{
  char machines[MAX_CONTENT_LENGTH],
       machine[MAX_NAME_LENGTH],
          remote_command[MAX_LINE_LENGTH],
          scope[2];
  datum local_key,
          local_content;
  int  marker;

  local_key.dptr = "net/remotes";
  local_key.dsize = strlen ("net/remotes") + 1;
  local_content = fetch (local_key);
  if (local_content.dptr == 0)
  {  printf ("%s: can't find 'net/remotes' entry in database.0, HOST_NAME);
     exit (1);
  }
  strncpy (machines, local_content.dptr, local_content.dsize);
  for (marker = 0;
       machines[marker] != '  '; )
  {
     for ( ; machines[marker] == ' '; ++marker);
```

```
    sscanf (&machines[marker], "%s", machine);
    for ( ; (machines[marker] != ' ') &&
            (machines[marker] != '  '); ++marker);

    strcpy (remote_command, "rsh ");
    strcat (remote_command, machine);
    strcat (remote_command, " '");
    strcat (remote_command, RELATIVE_GDBM_ACCESS_PATH);
    strcat (remote_command, " ");
    strcat (remote_command, database_name);
    strcat (remote_command, " ");
    strcat (remote_command, command);
    strcat (remote_command, " ");
    scope[0] = LOCAL;
    scope[1] = '  ';
    strcat (remote_command, scope);
    strcat (remote_command, " ");
    strncat (remote_command, key.dptr, key.dsize);
    strcat (remote_command, " ");
    strncat (remote_command, content.dptr, content.dsize);
    strcat (remote_command, "' &");
    system (remote_command);
  }
}
```

```
/*************************************************************************

   This is module netdb.c which provides any program with the read
   operation necessary to directly access the local file of a special
   global database. It is part of thesis work done by Lt. W. Cory Frank
   during summer and fall quarter 1987 under the title, "UNIX Based
   Programming Tools for Locally Distributed Network Applications."

*************************************************************************/

#include "netdb.h"
#include "netlocal.h"

db_fetch (key_string, content_string)
  char key_string[],
       content_string[];
{
  datum key,
        content;

  db_init ();

  key.dptr = key_string;
  key.dsize = strlen (key_string) + 1;
  content = fetch (key);
  if (content.dptr == 0)
    return (-1);
  else
  { strncpy (content_string, content.dptr, content.dsize);
    return (0);
  }
}


db_init()
{
  char database_path [64];

  strcpy (database_path, DATABASE_ROOT_PATH);
  strcat (database_path, "net");
  if (dbminit (database_path) < 0)
  { printf ("%d: %d: ", HOST_NUMBER, getpid());
    perror ("Can't init database in db_init");
    exit (1);
  }
}
```

```
/*****************************************************************************

    This is an IRIS-2400 program called netdraw.c. It is an adaptation
    of draw.c to be used in demonstrating the distributed network programs
    developed summer and fall quarter 1987 by Lt. W. Cory Frank.
    It is a part of thesis work under the title, "UNIX Based Programming
    tools for Locally Distributed Network Applications."
    It shows an example of line rubberbanding via polling with an option
    to send and receive mouse data from remote processes (same program).

*****************************************************************************/

#include "gl.h"
#include "device.h"
#include "netglobal.h"
#include "netlocal.h"
#include "netmsgtypes.h"
#include "netipc.h"

#define WXMAX 50.0
#define WYMAX 37.5

/*** These external variables are from netipc.c
***/
extern long local_node_addr,
            cntrl_node_addr;

main(argc, argv)
  int  argc;
  char *argv[];
{
  /*** Make some object names
  ***/
  Object singleline;
  Object accumulative;

  /*** make some tag names
  ***/
  Tag movetag,drawtag;

  /*** temp x,y coords of line
  ***/
  float x,y;

  /*** remote mouse buttons
  ***/
```

```
long mouse1, mouse2, mouse3,
        mousex, mousey,
        last_mouse1, last_mouse2, last_mouse3,
        last_mousex, last_mousey;

/*** for network use
***/
long client_node_addr [NUMBER_OF_HOSTS + 1];
int  count;
char data_buff[24];
msg_buff msg;

/*** for global database use
***/
char key_string[32],
        content_string[32];

/*** writemask spot
***/
Colorindex wmask;


/***  Initialize a shared memory segment and
        connect to the network controller.
***/
shd_mem_init ();


/*** Advertise the local node address.
***/
strcpy (key_string, "netdraw/");
strcat (key_string, HOST_NAME);
sprintf (content_string, "%d", local_node_addr);
cntrl_db_store (key_string, content_string);


/*** Get node addresses of client processes from command line.
***/
for (count = 1;
    (count <= NUMBER_OF_HOSTS) && (count < argc);
    ++count)
{  if (strcmp (argv[count], HOST_NAME) == 0)
   {  printf ("%d: %d: Sorry, you can't list your own host as a client.0,
              HOST_NUMBER, getpid());
     exit (0);
   }
```

```
  strcpy (key_string, "netdraw/");
  strcat (key_string, argv[count]);
  while (db_fetch (key_string, content_string) < 0)
  {  sleep (2);
  }
  sscanf (content_string, "%d", &client_node_addr[count]);
}

/*** initial values for x,y
***/
x=WXMAX/2.0;
y=WYMAX/2.0;

/*** initialize the IRIS system
***/
ginit();

/*** put the IRIS into double buffer mode
***/
doublebuffer();

/*** configure the IRIS (means use the above command settings)
***/
gconfig();

/*** enable all the bit planes for writing
    sets to 2**(getplanes()) minus one (all bits on)
***/
wmask=((1<<getplanes()) -1);
writemask(wmask);

/*** Full screen viewport
***/
viewport(0,1023,0,767);

/*** orthogonal projection 2D for the world coord sys
***/
ortho2(0.0,50.0,0.0,37.5);

/*** set initial value and range limit for mouse
***/
setvaluator(MOUSEX,511,0,1023);
setvaluator(MOUSEY,384,0,767);

/*** open the single line object
***/
```

```
singleline=genobj();

makeobj(singleline);
  /*** make the lines 3 pixels thick
  ***/
  linewidth(3);
  movetag=gentag();
  maketag(movetag);
  move2(WXMAX/2.0,WYMAX/2.0);
  drawtag=gentag();
  maketag(drawtag);
  draw2(WXMAX/2.0,WYMAX/2.0);
closeobj();

/*** open an empty object called accumulative
***/
accumulative=genobj();

makeobj(accumulative);
  /*** make the lines 3 pixels thick
  ***/
  linewidth(3);
  move2(x,y);
closeobj();

/*** turn on the default cursor
***/
setcursor(0,RED,wmask);

/*** attach the cursor to the mouse
***/
attachcursor(MOUSEX,MOUSEY);

/*** turn the cursor on
***/
curson();

while(TRUE)
{
  /*** obtain incoming remote data if any
  ***/
  if (net_poll (cntrl_node_addr, REFERENCE_ONLY, &msg) == 0)
  {  bcopy (msg.ref_data_ptr + 4, &mouse1, 4);
       bcopy (msg.ref_data_ptr + 8, &mouse2, 4);
       bcopy (msg.ref_data_ptr + 12, &mouse3, 4);
       bcopy (msg.ref_data_ptr + 16, &mousex, 4);
```

```
      bcopy (msg.ref_data_ptr + 20, &mousey, 4);

      ack_msg (&msg);

   last_mouse1 = mouse1;
      last_mouse2 = mouse2;
      last_mouse3 = mouse3;
      last_mousex = mousex;
      last_mousey = mousey;

   setvaluator (MOUSEX, mousex, 0, 1023);
      setvaluator (MOUSEY, mousey, 0, 767);

      x = (WXMAX / 1023.0) * mousex;
      y = (WYMAX /  767.0) * mousey;

 }
 else
 {  /*** read and save local data
   ***/
   mouse1 = getbutton (MOUSE1);
   mouse2 = getbutton (MOUSE2);
   mouse3 = getbutton (MOUSE3);
   mousex = getvaluator (MOUSEX);
   mousey = getvaluator (MOUSEY);

   /*** Use and send local data to clients if there was a change.
   ***/
   if ((mouse1 != last_mouse1) ||
         (mouse2 != last_mouse2) ||
         (mouse3 != last_mouse3) ||
         (mousex != last_mousex) ||
         (mousey != last_mousey))
   {  last_mouse1 = mouse1;
         last_mouse2 = mouse2;
         last_mouse3 = mouse3;
         last_mousex = mousex;
         last_mousey = mousey;

         x = (WXMAX / 1023.0) * mousex;
         y = (WYMAX /  767.0) * mousey;

      for (count = 1;
            (count <= NUMBER_OF_HOSTS) && (count < argc);
            ++count)
      {  bcopy (&client_node_addr[count], &data_buff[0], 4);
```

```
        bcopy (&mouse1, &data_buff[4], 4);
        bcopy (&mouse2, &data_buff[8], 4);
        bcopy (&mouse3, &data_buff[12], 4);
        bcopy (&mousex, &data_buff[16], 4);
        bcopy (&mousey, &data_buff[20], 4);
        make_msg (data_buff, 24,
                    FORWARD_MSG, REFERENCE_ONLY, &msg);
        if (net_send (cntrl_node_addr, &msg,
              WAIT_FOR_ACKNOWLEDGE, DISREGARD_ACKNOWLEDGE) < 0)
        {  printf ("%d: %d: error calling net_send0,
                     HOST_NUMBER, getpid ());
        }
      }
    }
  }

/*** quit if the middle mouse button is pushed
***/
if (getbutton (MOUSE2))
{
     break;
}

/*** should we start a new line?
***/
if(mouse1)
{
  /*** start a new line at this spot
       dont add the dragged line to accumulative
      ***/
  editobj(singleline);
    objreplace(movetag);
    move2(x,y);
    objreplace(drawtag);
    draw2(x,y);
  closeobj();

     /*** add a move2 command to the accumulative guy
     ***/
     editobj(accumulative);
       move2(x,y);
     closeobj();
}

/*** should we add a line to the accumulative object?
***/
```

```
  if(mouse3)
  {
    /*** add line to the accumulative object
       ***/
    editobj(accumulative);
      draw2(x,y);
    closeobj();

    /*** write x,y over movetag of single line object
       ***/
    editobj(singleline);
      objreplace(movetag);
      move2(x,y);
    closeobj();

  }


  color(CYAN);
  clear();

  /*** move the draw part of single line
  ***/
  editobj(singleline);
    objreplace(drawtag);
    draw2(x,y);
  closeobj();

  /*** draw the single line
  ***/
  color(BLACK);
  callobj(singleline);

  /*** draw the accumulative lines buffer
  ***/
  callobj(accumulative);

  /*** change the buffers
  ***/
  swapbuffers();

}

/*** perform some clean-up by setting the graphics to black
***/
color(BLACK);
```

```
  clear();
  swapbuffers();
  clear();
  textinit();
  swapbuffers();
  finish();

  /*** graphics exit
  ***/
  gexit();

  /*** close the connection to the network controller
  ***/
  shd_mem_detach ();
}


/*** Store operations for global database. No lock protection provided!
***/
cntrl_db_store (key_string, content_string)
  char key_string[],
        content_string[];
{
  char syscmd[128];

  strcpy (syscmd, "/work/frank/network/dbcntrl net s g ");
  strcat (syscmd, key_string);
  strcat (syscmd, " ");
  strcat (syscmd, content_string);
  strcat (syscmd, " &");
  system (syscmd);
}
```

```
/*************************************************************************

    This is program netaccess.c which provides a format for user
    storage of subnet data for use in the related logical network.
    This is part of thesis work done by W. Cory Frank during summer and
    fall quarter 1987 under the title, "UNIX Based Programming Tools for
    Locally Distributed Network Applications." This program shows how the
    recommended interface should respond but is not linked to actual
    network operations yet. This cannot be done until the related
    'network server' program is implemented. The program can be expanded
    easily to handle any network related user request.

*************************************************************************/

#include "netglobal.h"
#include "netlocal.h"
#include "netipc.h"
#include "netdb.h"
#include "netmsgtypes.h"

/*** These external variables are from netipc.c
***/
extern long local_node_addr,
            cntrl_node_addr;
main ()
{
  char usr_cmd [80];


  /*** to be used when actually linked to network operations
  shd_mem_init ();
  ***/


  strcpy (usr_cmd, "  ");
  while ((usr_cmd[0] != 'q') && (usr_cmd[0] != 'Q'))
  {  printf ("command? ");
     scanf ("%s", usr_cmd);
     switch (usr_cmd[0])
     {  case 'm':
          case 'M': make_subnet ();
                    break;
          default: break;
     }
  }
}
```

```
make_subnet ()
{
  subnet_data subnet;
  int       source,
            target;

  get_subnet_template (subnet.template);
  get_num_resources (&subnet.num_resources);
  init_subnet_data (&subnet.res_names[0][0],
              &subnet.res_templates[0][0], &subnet.res_machines[0][0],
            &subnet.connect_masks[0][0], subnet.num_resources);
  display_subnet_data (subnet);
  for (source = 0;
        source < subnet.num_resources;
        source = source + 1)
  {  get_resource_name (subnet.res_names, subnet.res_names[source], source);
     get_template_machine_pair (subnet.res_templates[source],
                                subnet.res_machines[source]);
     display_subnet_data (subnet);
  }
  for (source = 0;
        source < subnet.num_resources;
        source = source + 1)
  {  for (target = source + 1;
           target < subnet.num_resources;
           target = target + 1)
     {  get_connection_info (&subnet.connect_masks[source][target],
                     source, target);
        display_subnet_data (subnet);
     }
  }
  /*** to be used when actually linked to network operations
  store_subnet_data (&subnet, sizeof (subnet));
  ***/
}



get_subnet_template (template)
  char template[];
{
  int valid_input;

  for (valid_input =  -1;
      valid_input == -1;
        valid_input =  is_new_key (template) )
```

```
  {  printf ("0nput the new subnet template name:   ");
     scanf ("%s", template);
  }
}



is_new_key (template)
  char template[];
{
  return (0);
}



get_num_resources (num_resources_ptr)
  int *num_resources_ptr;
{
  int valid_number;
  char num_resources_string [80];

  for (valid_number =  -1;
         valid_number == -1; )
  {  printf ("0nput the total number of resource");
     printf (" modules within the subnet:   ");
     scanf ("%s", num_resources_string);
     sscanf (num_resources_string, "%d", num_resources_ptr);
     if ((*num_resources_ptr > 0) &&
           (*num_resources_ptr <= MAX_SUBNET_RESOURCES))
        valid_number = 1;
  }
}



init_subnet_data (name_ptr, template_ptr, machine_ptr,
              mode_ptr, num_resources)
  char *name_ptr,
         *template_ptr,
         *machine_ptr,
         *mode_ptr;
  int  num_resources;
{
  int  offset,
         max_offset;
```

```
  max_offset = MAX_SUBNET_RESOURCES * (MAX_NAME_LENGTH + 1) - 1;
  for (offset = 0;
        offset <= max_offset;
        offset =  offset + 1)
  {  *(name_ptr + offset) = '-';
    *(template_ptr + offset) = '-';
    *(machine_ptr + offset) = '-';
  }
  for (offset = 0;
        offset < max_offset;
        offset = offset + MAX_NAME_LENGTH + 1)
  {  *(name_ptr + offset) = '?';
    *(template_ptr + offset) = '?';
    *(machine_ptr + offset) = '?';
  }
  for (offset =  MAX_NAME_LENGTH;
        offset <= max_offset;
        offset =  offset + MAX_NAME_LENGTH + 1)
  {  *(name_ptr + offset) = ' ';
    *(template_ptr + offset) = ' ';
    *(machine_ptr + offset) = ' ';
  }

  max_offset = MAX_SUBNET_RESOURCES * (MAX_SUBNET_RESOURCES + 1) - 1;
  for (offset =  0;
        offset <= max_offset;
        offset =  offset + 1)
  {  *(mode_ptr + offset) = '?';
  }
  for (offset = 0;
        offset < max_offset;
        offset = offset + MAX_SUBNET_RESOURCES + 2)
  {  *(mode_ptr + offset) = 'x';
  }
  for (offset =  num_resources;
        offset <= max_offset;
        offset =  offset + MAX_SUBNET_RESOURCES + 1)
  {  *(mode_ptr + offset) = ' ';
  }
}


display_subnet_data (subnet)
  subnet_data   subnet;
{
```

```
  int source,
     target;

  printf ("0atrix for subnet template '%s':0, subnet.template);
  printf ("------------------------------- target:0);
  printf ("#__source____template__machine___");
  for (target = 0;
        target < subnet.num_resources;
        target = target + 1)
  {  printf ("%2d_", target + 1);
  }
  printf ("0);
  for (source = 0;
        source < subnet.num_resources;
        source = source + 1)
  {  printf ("%-3d%-10s%-10s%-10s ", source + 1, subnet.res_names[source],
              subnet.res_templates[source], subnet.res_machines[source]);
    for (target = 0;
          target < subnet.num_resources;
          target = target + 1)
    {  printf ("%c  ", subnet.connect_masks[source][target]);
    }
    printf ("0);
  }
  for ( ; source < 15; ++source) { printf ("0); }
}



get_resource_name (res_names, name, source)
  char res_names[][MAX_NAME_LENGTH + 1],
        name[];
  int  source;
{
  int  valid_resource_name;

  for (valid_resource_name =  -1;
      valid_resource_name == -1;
      valid_resource_name =  is_new_resource (res_names, source) )
  {  printf ("0nput name for subnet source number%3d:   ", source + 1);
    scanf ("%s", name);
  }
}
```

```
is_new_resource (res_names, source)
  char res_names[][MAX_NAME_LENGTH + 1];
  int  source;
{
  return (0);
}



get_template_machine_pair (template, machine)
  char template[],
         machine[];
{
  int  valid_template,
       valid_machine;

  for (valid_template =  -1;
       valid_template == -1;
       valid_template =  res_template_at_machine (template, machine) )
  {  printf ("Input its associated source template  :   ");
     scanf ("%s", template);
     for (valid_machine =  -1;
          valid_machine == -1;
          valid_machine =  is_valid_machine (machine) )
     {  printf ("Input source/template machine location:   ");
        scanf ("%s", machine);
     }
  }
}



res_template_at_machine (template, machine)
  char template[],
         machine[];
{
  return (0);
}



is_valid_machine (machine)
  char machine[];
{
  return (0);
}
```

```
get_connection_info (mode_ptr, source, target)
  char *mode_ptr;
  int  source,
         target;
{
  int  valid_mode;
  char input[1];

  for (valid_mode =  -1;
      valid_mode == -1;
      valid_mode =  is_valid_mode (*mode_ptr, target) )
  {  printf ("0nput connection mode from source #%d to target #%d: ",
              source + 1, target + 1);
    scanf ("%s", input);
    *mode_ptr = input[0];
  }
}



is_valid_mode (mode, target)
  char mode;
  int  target;
{
  return (0);
}

/*** to be used when actually linked to network operations
store_subnet_data (subnet_ptr, subnet_size)
  subnet_data  *subnet_ptr;
  int  subnet_size;
{
  char data_buff[MAX_HARD_COPY_SIZE];
  msg_buff msg;

  bcopy (subnet_ptr, data_buff, subnet_size);
printf ("%x: bcopy done0, local_node_addr);
  make_msg (data_buff, subnet_size, DB_STORE_GLOBAL,
                    REFERENCE_ONLY, &msg);
printf ("%x: message made0, local_node_addr);
  if (net_send (cntrl_node_addr, &msg, DISREGARD_ACKNOWLEDGE,
                              WAIT_FOR_ACKNOWLEDGE) < 0)
    printf ("%x: subnet storage msg not sent or action denied by cntrl0,
            local_node_addr);
}
***/
```

```
/*************************************************************************

   This is a UNIX V program called rmvshmids.c which removes residual
   shared memory IDs from the system (only those created by the owner
   of a process from this program). It is essential that they do
   remain longer than necessary as this would be a waste of memory.

*************************************************************************/

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

main ()
{
  int count;
  struct shmid_ds *buf;

  for (count = 0;
        count <= 10000;
        ++count)
  {  if (shmctl (count, IPC_RMID, buf) == 0)
        printf ("%d: shared memory id removed0, count);
  }
}
```

```
/*************************************************************************

   This is header netglobal.h for general use within the entire network.

*************************************************************************/

#define MAX_NAME_LENGTH      9
#define MAX_PATH_LENGTH     80
#define MAX_CONTENT_LENGTH  80
#define MAX_LINE_LENGTH     80

#define NUMBER_OF_HOSTS 3
#define HOSTS_IN_ORDER  "iris1 iris2 iris3"
```

```
/*****************************************************************************

    This is header netlocal.h for use on a specific host within the network.

*****************************************************************************

#define HOST_NUMBER  2
#define HOST_NAME    "iris2"
#define REMOTE_NAMES "iris1 iris3"
```

```
/*************************************************************************

   This is header netipc.h for use with UNIX V shared memory IPC.

*************************************************************************/

#define ALL_NODE_ADDR    0

#define REFERENCE_ONLY     0
#define REFERENCE_AND_COPY 1

#define MAX_HARD_COPY_SIZE    768 /*** bytes; smaller than message space ***/

#define MSG_ID_TO_ADDR_MASK   0x7FFFE000
#define MSG_ID_TO_HOST_MASK   0x70000000

#define RETURN_IF_NO_ACKNOWLEDGE 0
#define WAIT_FOR_ACKNOWLEDGE     1
#define DISREGARD_ACKNOWLEDGE    2

typedef struct
{  char hard_copy_data[MAX_HARD_COPY_SIZE],
        *ref_msg_ptr,
        *ref_data_ptr;
  long hard_copy_id,
        *ref_id_ptr;
  int  msg_mode,
        msg_type,
        sent_link,
        rcvd_link,
        hard_copy_size,
        ref_data_size;
} msg_buff;

#define MAX_MSG_ID   0x1FFF   /*** 13 low order bits that are
                              unused in a physical node address ***/

#define SHD_MEM_CNTRL_LINK    0

#define SHD_MEM_NUM_SEGMENTS       16 /*** logical divisions made in
                                      a single physical segment ***/
#define SHD_MEM_SPACE_SIZE       2048 /*** bytes long ***/
#define SHD_MEM_ADT_SPACE_OFFSET     0 /*** bytes from start of segment ***/
#define SHD_MEM_ADT_SPACE_SIZE     512 /*** bytes long ***/
#define SHD_MEM_MSG_SPACE_OFFSET   512 /*** bytes from start of segment ***/
#define SHD_MEM_MSG_SPACE_SIZE    1024 /*** bytes long ***/
```

```
#define SHD_MEM_LINK_SPACE_OFFSET 1536 /*** bytes from start of segment ***/
#define SHD_MEM_LINK_SPACE_SIZE   128 /*** bytes long ***/
#define SHD_MEM_MAX_LINKS         10 /*** (link space size) % (12 bytes) ***/

#define SHD_MEM_MSG_ID_OFFSET       0 /*** bytes ***/
#define SHD_MEM_MSG_TYPE_OFFSET     4 /*** bytes ***/
#define SHD_MEM_MSG_DATA_SIZE_OFFSET 8 /*** bytes ***/
#define SHD_MEM_MSG_DATA_OFFSET     12 /*** bytes ***/
#define SHD_MEM_MSG_NON_DATA_BYTES  12 /*** bytes ***/

typedef struct
{  long remote_node_addr,
      remote_segment_offset,
      *remote_msg_id_ptr,
      *remote_msg_offset_ptr,
         *remote_ack_id_ptr,
         *local_msg_id_ptr,
         *local_msg_offset_ptr,
      *local_ack_id_ptr;
} shd_mem_link_info;

typedef struct
{  char host_name[16];
  int  port,
         socket;
  long sent_msg_id,
         ack_msg_id;
} inter_host_info;
```

```
/*****************************************************************************

    This is header netdb.h for use with global database operations.

*****************************************************************************/

#include <bsd/dbm.h>
#include "netglobal.h"
#include "netlocal.h"

#define  DATABASE_ROOT_PATH "/work/frank/do_not_copy!/"
#define  RELATIVE_GDBM_ACCESS_PATH "network/dbcntrl"

#define DBM_CREATE 'c'
#define DBM_STORE  's'
#define DBM_FETCH  'f'
#define DBM_DELETE 'd'

#define LOCAL  'l'
#define GLOBAL 'g'

#define MAX_SUBNET_RESOURCES 15

typedef struct
{  char template     [MAX_NAME_LENGTH + 1],
        res_names     [MAX_SUBNET_RESOURCES] [MAX_NAME_LENGTH + 1],
        res_templates [MAX_SUBNET_RESOURCES] [MAX_NAME_LENGTH + 1],
        res_machines  [MAX_SUBNET_RESOURCES] [MAX_NAME_LENGTH + 1],
        connect_masks [MAX_SUBNET_RESOURCES] [MAX_SUBNET_RESOURCES + 1];
  int  num_resources;
}  subnet_data;
```

```
/*****************************************************************************

   This is header netmsgtypes.h which holds all the message types
   for use within the logical network.

*****************************************************************************/

#define ANY_MSG_TYPE  0

#define SHD_MEM_REQ_CONN  1
#define SHD_MEM_ACK_CONN  2
#define SHD_MEM_REQ_BREAK 3
#define SHD_MEM_ACK_BREAK 4

#define SUBNET_CREATE    10
#define SUBNET_DELETE    11
#define NETWORK_SHUTDOWN 12

#define FORWARD_MSG     20
#define REMOTE_MSG_ACK  21

#define DB_STORE_GLOBAL 30
```

LIST OF REFERENCES

[1] *UNIX Programmer's Manual* (Silicon Graphics, Inc., Mountain View, California, 1986), IB .

[2] Rochkind, M. J., *Advanced UNIX Programming* (Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1985).

[3] III, A. N. Cooper and Kearns, J. P., "Real-Time Distributed Control with Asynchronous Message Reception," Proceedings of the Real-Time System Symposium, 1985, IEEE Computer Society, December, 1985.

[4] Leffler, S. J., Fabry, R. S., Joy, W. N., Lapsley, P., Miller, S., and Torek, C., *An Advanced 4.3BSD Interprocess Communication Tutorial* (Computer Systems Research Group, Department of Electrical Engineering and Computer Science, University of California, Berkeley, California, 1986).
Communications of the ACM 17, 549 - 557 (October, 1974).

[6] Parnas, D. L., "On the Criteria To Be Used in Decomposing Systems into Modules," *Communications of the ACM 15*, 1053 - 1058 (December, 1972).

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Chief of Naval Operations Director, Information Systems (OP-945) Navy Department Washington, D.C. 20350-2000	1
2.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22304-6145	2
3.	Library, Code 0142 Naval Postgraduate School Monterey, California 93943-5002	2
4.	Department Chairman, Code 52 Department of Computer Science Naval Postgraduate School Monterey, California 93943-5000	2
5.	Curriculum Officer, Code 37 Computer Technology Naval Postgraduate School Monterey, California 93943-5000	1
6.	Associate Professor Michael J. Zyda, Code 52Zk Computer Science Department Naval Postgraduate School Monterey, California 93943-5000	2
7.	Major Richard A. Adams, USAF Code 52Ad Computer Science Department Naval Postgraduate School Monterey, California 93943-5000	1
8.	Lt. W. Cory Frank, USN Naval Security Group Detachemnt Corry Station Pensacola, Florida 32511-5100	3

322 - 496

Thesis
F78225 Frank
c.1 UNIX based programming
tools for locally distri-
buted network application.

Thesis
F78225 Frank
c.1 UNIX based programming
tools for locally distri-
buted network application.

AMERICAN BINDERY
TOPEKA KANSAS

thesF78225
UNIX based programming tools for locally
3 2768 000 76868 3
DUDLEY KNOX LIBRARY